5

NOTE DI SOFTWARE

ARCHIVIO STORICO
ASSOCIAZIONE POZZO DI MIELE
FDDM-111
RNSW-103

UNIVERSITA'
DEGLI STUDI DI MILANO
ISTITUTO DI CIBERNETICA

Honeywell
Honeywell Information Systems Italia

Note di software.
è un bollettino trimestrale, edito a cura del Centro di Ricerca e Progettazione della Honeywell Information Systems Italia e dell'Istituto di Cibernetica dell'Università di Milano.

Note di software.
intende costituire uno strumento per la rapida e informale circolazione di informazioni, idee ed esperienze nell'area della tecnologia del software. In particolare, intende stimolare il dibattito sulle metodologie orientate alla diminuzione del rapporto costo/qualità dei programmi, e sugli strumenti atti ad automatizzare l'attività di produzione del software in ogni suo aspetto.

La collaborazione a **Note di software** è libera.
In particolare, si sollecitano contributi nella forma di brevi monografie o bibliografie essenziali commentate sui seguenti temi: tecniche e strumenti per la programmazione strutturata e modulare, metodologie e strumenti per il testing e il debugging dei programmi, documentazione del software, aspetti organizzativi e gestionali nella costruzione di sistemi software di grandi dimensioni.

Chi desiderasse pubblicare un contributo è pregato di prendere contatto con il Comitato di Redazione (HISI - Via ai Laboratori Olivetti - Pregnana Milanese), o di inviare direttamente il dattiloscritto in triplice copia.
La preparazione del testo nel formato adatto alla fotoriproduzione è a carico degli autori, ai quali verranno comunicate le norme redazionali.
I contributi non dovrebbero superare le quattromila parole.

I lavori accettati per la pubblicazione non vengono revisionati da un comitato tecnico. Pertanto, ogni contributo pubblicato riflette le opinioni personali dei suoi autori, e non necessariamente quelle del Comitato di Redazione.
Eventuali revisioni saranno effettuate di comune accordo fra il Comitato di Redazione e i proponenti.

Note di software.
viene distribuito a chi ne faccia richiesta alla redazione.

Direttore responsabile: Paolo Ghelfi
Comitato di Redazione: Roberto Polillo - Wanda Anselmetti
Redazione: Franco Soresini

NOTE DI SOFTWARE

5

Gennaio - Marzo 1978

FPDM-111

Indice:

QUESTO NUMERO . . . . pag. 1

CONTRIBUTI TECNICI:

- OVERVIEW OF 'WELLMADE' DESIGN METHODOLOGY, di D. L. Boyd e A. Pizzarello " 2
- SECOB: UN PRECOMPILATORE COBOL INTEGRATO PER LA PROGRAMMAZIONE STRUTTURATA, di S. Antocicco, C. Barassi, M. Maiocchi, P. Marighella " 10
- GENERAZIONE AUTOMATICA DI UNITA' DI PROGRAMMA: UN APPROCCIO, di E. Fischetti e G. Petraglia " 20
- LA GENERAZIONE AUTOMATICA DEI DATI DI TEST, di G. Ferraro e D. Marini " 27
- UNO STRUMENTO PER LA GESTIONE DI ARCHIVI DI TEST, di A. Jemoli " 33
- SPECIALIZZAZIONE DI UN COMMAND LANGUAGE IN AMBIENTE BATCH REMOTO, di G. Galdi, G. Petraglia, G. Vildacci " 38

BIBLIOGRAFIA:

- IL LINGUAGGIO PASCAL NELLA LETTERATURA, di G. Castelli " 44

AVVENIMENTI:

- SUCCESSFUL SOFTWARE MANAGEMENT TECHNIQUES CONFERENCE, di A. Cicu e P. Ghelfi " 52

IL SEGNALIBRO " 56

Autorizzazione del Tribunale di Milano n. 87 del 23 febbraio 1977

QUESTO NUMERO . . .

Il quinto numero di NOTE DI SOFTWARE raccoglie sei contributi su alcuni aspetti centrali dell'ingegneria del software: dalle metodologie di progettazione, all'attività di testing, a quella di programmazione.

Nel primo lavoro ("Overview of WELLMADE Design Methodology"), Boyd e Pizzarello introducono, nelle sue linee generali, WELLMADE, una metodologia di "software design" che costituisce una sintesi di risultati recenti della ricerca sulla programmazione e dell'esperienza acquisita nei laboratori Honeywell di Phoenix. WELLMADE, basandosi sull' "approccio costruttivo" proposto da Dijkstra e sul concetto di tipo di dato astratto, definisce un proprio linguaggio di specifica, che sarà oggetto di future pubblicazioni su NOTE DI SOFTWARE.

Antocicco, Barassi, Maiocchi e Marighella ("SECOB: un precompilatore Cobol integrato per la programmazione strutturata") presentano SECOB, un precompilatore Cobol che mette a disposizione del programmatore le strutture di controllo della programmazione strutturata, e impone al programma una organizzazione gerarchica, in cui le istruzioni di ogni modulo sono fisicamente vicine alla definizione dei dati da esse utilizzati. Rilevante, in SECOB, è il fatto che esso fornisce una lista di compilazione integrata con quella del compilatore ospite, fornendo così al programmatore un documento unico, contenente tutte le informazioni relative al programma.

Il lavoro di Fischetti e Petraglia ("Generazione automatica di unità di programma: un approccio") propone un approccio metodologico alla progettazione di procedure EDP, che permette la prova della struttura portante della procedura stessa (JCL e "scheletri" dei vari programmi) già in fase di "system design", e descrive uno strumento, realizzato sul Livello 62, per la generazione automatica di tali "scheletri" di programma.

Altri due lavori trattano problemi connessi all'attività di prova di prodotti software, vista in due momenti diversi. Nel primo ("La generazione automatica di dati di test"), Ferraro e Marini analizzano il problema della individuazione di un insieme di dati di prova che permettano di esercitare tutte le ramificazioni di un programma, e descrivono uno strumento (PATHPRED) che genera automaticamente, data in ingresso la descrizione di un "cammino di controllo", il "predicato di test" associato (una relazione, cioè, che deve essere soddisfatta dai dati di ingresso al programma da provare, affinchè tale cammino venga esercitato). Nel secondo ("Uno strumento per la gestione di archivi di test"), Jemoli descrive uno strumento che permette di gestire, in modo automatizzato, la selezione, la preparazione e la esecuzione di "catene" di test già realizzati, archiviate in una libreria opportunamente standardizzata.

Il lavoro di Galdi, Petraglia e Vildacci ("Specializzazione di un command language in ambiente batch remoto") indica come la disponibilità di un JCL altamente evoluto permetta di realizzare una gestione completa di programmi da remoto, anche in assenza di strumenti di Remote Job Entry. L'articolo descrive, in particolare, un sistema (RBE: Remote Batch Entry) realizzato sul Livello 62.

Castelli, infine, presenta una bibliografia aggiornata sul linguaggio di programmazione Pascal.

I lettori troveranno, in questo fascicolo di NOTE DI SOFTWARE, un inserto con un questionario. E' lo strumento che abbiamo ritenuto più idoneo per un "dialogo" con i lettori della rivista. Confidiamo di ricavare, dalle risposte, informazioni relative agli interessi, alle attività dei lettori, e alla possibilità di operare per una sempre maggior diffusione di competenze nel settore.

La Redazione

CONTRIBUTI TECNICI

OVERVIEW OF 'WELLMADE' DESIGN METHODOLOGY

D.L.Boyd(°), A. Pizzarello(+)

1. INTRODUCTION

The WELLMADE methodology has been developed as an aid to higher quality software design. It has been pointed out in several studies [BOEH75, MILL76] that the design stage is the most critical to produce high quality software. However, very often there is high confusion in the very definition of design, and some of the statistics are difficult to understand because of the highly subjective interpretation of the word design.

The vision of software design activity which is at the origin of WELLMADE is that complex of activity which transforms the functional specifications (what the system is expected to do) together with the definition of system attributes such as target hardware or software (the boundary of the design activity) to produce the specifications of the "how" that mission will be accomplished by the system in the prescribed boundaries.

This definition of design corresponds with the common acception of the word in other engineering endeavors. It, clearly, does not limit design to establishing functional requirements and requires the detailed definition of the mechanisms necessary to accomplish the mission.
On the other hand, it may appear that all the software development activities are included in the above definition of design. However, this is not the intention of WELLMADE, which excludes actual coding of algorithm in the available programming language, and requires the definition of the algorithms in a design specification language. This is not different from the production of blue-prints as a result of any design activity, which specify all the necessary details of the final product, but are not the final product.
In software development a considerable lack of discipline and fuzzy boundaries between various stages of development (many important design decisions are made while debugging actual code) have produced an unhealthy situation of uncertainty about the behavior of the final product. Thus, a methodology to put order and discipline in the software development process appears to be strongly needed.

What is a methodology? We define a methodology as a complex of work procedures supported by scientific principles aimed to solve generalized design problems. These procedures are made effective by notation, tools, and management techniques. In this paper we give a brief description of which work procedures were chosen for WELLMADE to solve design problems. More detailed descriptions of WELLMADE can be found in [BOYD78] and in articles that will be published later in this magazine.

2. SCIENTIFIC PRINCIPLES

The basic concept of any software design methodology is to insure correctness.
The fundamental work on program correctness by Floyd and Hoare [FLOY67, HOAR69] is in fact at the basis of WELLMADE. However, WELLMADE relies on the idea of the constructive approach to program correctness [DIJK68, DIJK76, PIZZ77], which uses the proof ideas as a guide to derive correct programs from specifications. The process of demonstration that the program does, in fact, what it is supposed to do, is not possible if the "what" is not rigorously defined. But the program

(°) Corporate Computer Sciences Center, Honeywell Corporate Technology Center, Bloomington, Minnesota
(+) Honeywell Large Information Systems Division, Phoenix, Arizona

itself (possibly written in another programming language) could be considered a valid specification of "what", and this consideration appears to make the idea self-defeating. The solution proposed first by Floyd, and widely accepted today, is to specify what the program is supposed to do, in terms of static relationships between the program variables instead of specifying all its possible executions on the object machine. This idea of static specifications is the key of all the theory of proof of correctness and is useful also in itself, because by the process of describing statically the objective of a program, most (but not all) potential sources of errors are eliminated.

Another lesson learnt by the application of the principles of proof of correctness is that certain control structures used in the construction of programs from elementary structures are much more amenable to rigorous static specifications and correctness proofs than others. This observation is one of the principles of the so-called structured programming, where control structures as go to and data pointers are banned.

All the above principles derive from the idea of proof of correctness. However, practical applications of the proof of correctness appear manageable only for programs of limited size. This problem was solved first by Dijkstra [DIJK68] with the introduction of levels of abstraction in program design. In his first papers Dijkstra gave a correct, even if informal, formulation of the idea of abstraction, but later the idea was frequently misunderstood and incorrectly applied. A formalization of the idea of levels of abstraction was presented by Hoare in a difficult paper [HOAR72]. Subsequently, Wulf et al., in designing the language ALPHARD, and SRI methodology used levels of abstraction quite correctly [WULF76, ROUB77].
In WELLMADE, the concept of levels of abstraction is based on the use of abstract data types. A data type, as defined in WELLMADE, is completely specified when there are given:

Table 1

GENERAL DESIGN METHODOLOGY OVERVIEW

		SCIENTIFIC PRINCIPLES			
		STRUCTURE	STATIC SPECIFICATIONS	CONSTRUCTIVE METHODS	PROOF TECHNIQUES
DESIGN PROBLEMS	PROGRAMS	STRUCTURED CONSTRUCTS and PROCEDURAL ABSTRACTION	INPUT/OUTPUT SPECIFICATIONS	CONSTRUCTIVE APPROACH (DIJKSTRA) STEPWISE REFINEMENT	INDUCTIVE PROOFS (FLOYD, HOARE, DIJKSTRA)
	MACHINES	ABSTRACT DATA STRUCTURES and PERMANENT DATA	DATA TYPES and DATA INVARIANTS	HIERARCHICAL TYPE CONSTRUCTION	DATA REPRESENTATION MAPPING (HOARE, ALPHARD)
	CONCURRENCY	MONITORS (SHARED DATA)	SYNCHRONIZATION INVARIANTS	RESEARCH	RESEARCH

a)- a collection of values associated with objects of that type;

b)- a description of a relationship between these values, limiting the set of the acceptable values for the type, called the type invariant;

c)- the static functional specifications of all the permissible operations on objects of that type.

When types are defined as above, they actually specify a machine for which programs can be written and their correctness demonstrated. Thus by inventing appropriate types, sufficiently powerful abstract machines to permit to contain the size of programs are invented. Then, each one of the invented types are refined by creating a new machine, for which the operations of the type are programs written (and demonstrated correct) in terms of other types in a less powerful machine. The process is continued until the object machine is reached.
The actual use of this concept involves some technicalities which are not described in this paper.

3. DESIGN PROBLEMS

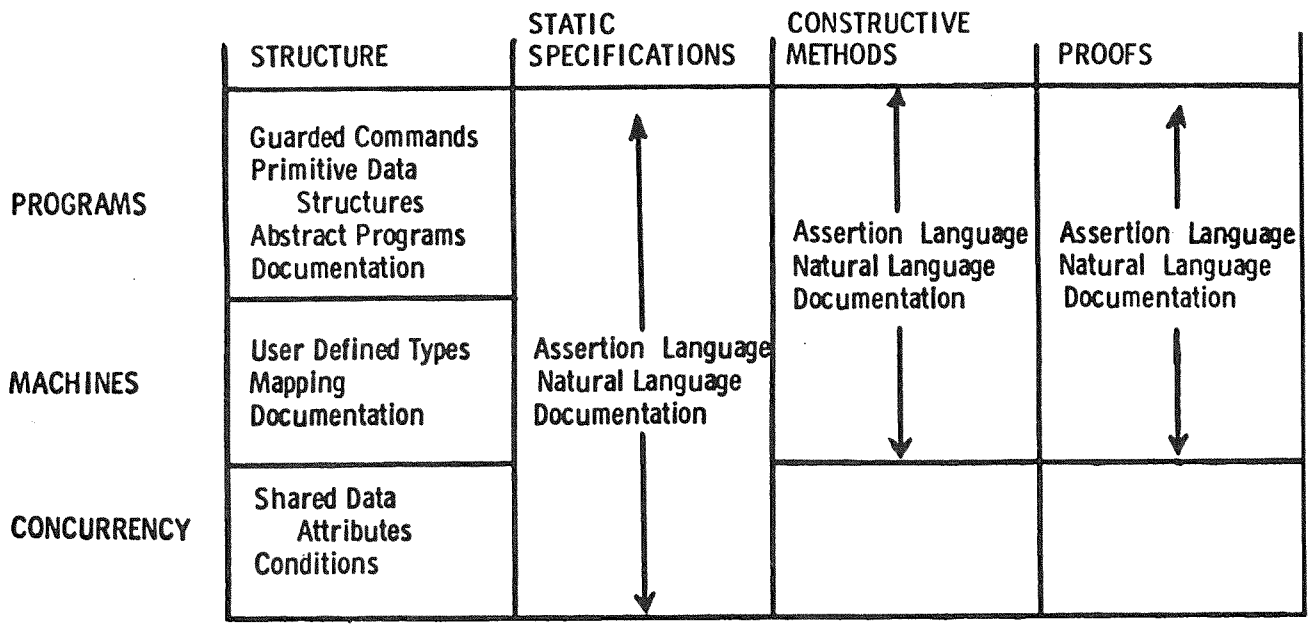
All software systems are certainly built up by programs. However, the idea of a single sequential program does not suffice to cover all the conceivable software systems.

For example, a payroll system is usually made up by an employee's file which contains permanent record of the information relative to all employees (the data base), and by several programs such as the one that prints the payslip, the one that calculates the income taxes, the one that updates the file by purging terminated employees and adding the new hired ones, etc. All these programs are disjoint from each other. The only commonality is that they all use the data base and, if they are correctly written, respect a set of common rules in dealing with it (called the data invariant). For example, if the employee's file is supposed to be sorted by pay number, all programs expect to find the file so sorted and no program will leave the file in a state violating the above rule.

This situation is perfectly realized by an abstract machine as defined above. Thus,

Table 2

WELLMADE NOTATION



abstract machines are not only an essential tool in the derivation of programs by levels of abstraction, but they also effectively model practical systems.

In the above model, all the programs of the system are intended to execute non-concurrently. How the non-concurrency is obtained was not said. Unfortunately in practical cases it is sometimes necessary to specify how the concurrency of execution can be obtained without undesired interferences. This problem of the correctness of concurrent programs is not yet satisfactorily solved in practical cases as the problem of design of correct programs or machines.

4. DESIGN PROCEDURES

The design process at every level begins with a requirements analysis. That is, the informal expression of the required capabilities must be transformed into static specifications which are sufficient to construct the programs for that level, and the abstract machine to support them.

These specifications are obtained by defining the data objects - and their types - necessary to describe the input and output specifications for the programs and the operations of the types.
The result of the abstract machine specification step must be program designs and specifications of a new environment, which suffice to show the correctness of the programs.
The design process then returns to requirements analysis, using as new requirements the specifications of those data types that are not supplied by the object machine.
The design then proceeds top-down. In WELLMADE, 'top-down design' means that:

- a program cannot be statically specified until all of its requirements are known (and understood),
- the program logic design cannot be derived until the static specifications are known, and
- the environment for static and programs design forms an abstract machine which in turn may generate requirements in the form of abstract data types to be designed as a lower level machine.

Table 3

WELLMADE NOTATION

MEANING	PROGRAM CONTROL P - NOTATION	DATA STRUCTURES V - NOTATION
Do Nothing	skip	abstract
Concatenation	S1; S2	V1; V2
Selection	if ... Bi→Si ▮ ... fi	V1 ▮ V2
Iteration	do ... Bi→Si ▮ ... od	V1 array
Assignment	V1, V2, ... Vn := E1, E2, ..., En	
where	Si represents program operations	
	Vi represents data structures	
	Bi represents Boolean expressions	
	Ei represents data structure expressions	

For example, in treating a sequential file it may be not needed to consider the fields in the records. Therefore the abstract data type 'record' is created, and the programs handling the file are written. The data type 'record' will be detailed in a lower level machine, where the functions used as instructions at the upper level are now realized in terms of the data type 'field', and so on.

In WELLMADE this technique is made formal by using the constructive approach, which combines Dijkstra's predicate transformers technique for the derivation of programs [DIJK76] with a stepwise refinement of procedures by using abstraction to name program's inner blocks which are later developed in detail.

The creation of new levels of abstract machines requires the demonstration of consistency of the new data representation. In most cases, if the appropriate techniques are used, the consistency is fairly easy to show. In other cases the demonstration is much more difficult. The necessary syntactic tools are supplied in the WELLMADE notation, and the appropriate methods to show consistency (similar to the techniques described in [HOAR72] and [WULF76]) are part of the methodology.

5. WELLMADE DOCUMENTATION STRUCTURE

Table 1 shows which technique is used in WELLMADE to solve the three generalized design problems (programs, machine, concurrency). It also shows which scientific principle (structure, static specifications, constructive approach, or a-posteriori proof of correctness) can be used.

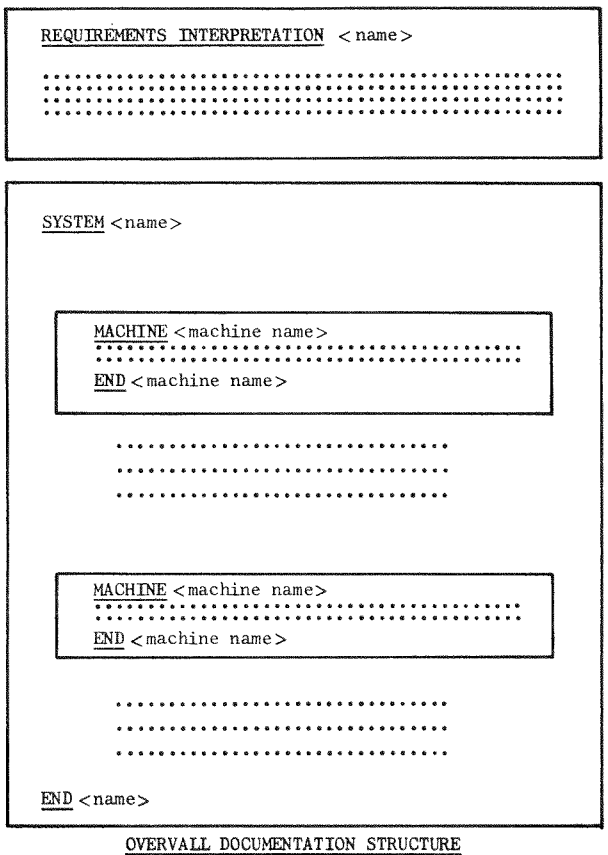
Three notations are necessary to represent the program design. These notations correspond to major design activities of the methodology. They are:

1. a specification language for representing assertions, predicates, and invariant conditions. This language, that in its most informal acception can be English, is necessary to specify "what" each program or type operation is supposed to do;
2. a program design language for representing detailed procedural logic and data structures of the design, and
3. a documentation format to support organization and management of the design process as well as to provide the necessary structural framework in which the data and algorithm notation can be used.

In Table 2 the notation used in WELLMADE is indicated. The details of the notation are not given here (see [HONE78]). In Table 3 the basic constructs of the algorithm and data definition language are shown. All of the design as defined in WELLMADE will result in documents organized according to a documentation scheme whose overall structure is shown in Table 4. The documentation is organized by machines and, as much as possible, each machine document is a self contained one. The requirements of the whole system are documented separately.

The system is designed by hierarchical construction of abstract machines; the documentation follows the same hierarchy (see Fig.1). Each machine document describes the processing of DATA by PROGRAMS according to the capabilities supplied by TYPES. Each type which is not primitive is, in turn, realized by another machine. In Fig.1, machine 4 and 5 are suppo-

Table 4



OVERVALL DOCUMENTATION STRUCTURE

sed to be constructed on primitive types only and therefore represent the bottom of the hierarchy.

In WELLMADE programs can make use of subprograms. In Fig.1 program P_{11} makes use of subprograms P_{12} and P_{13}. The latter in turn make use of P_{14} and P_{15}. Programs are described in appropriate chapters of the machine documents.

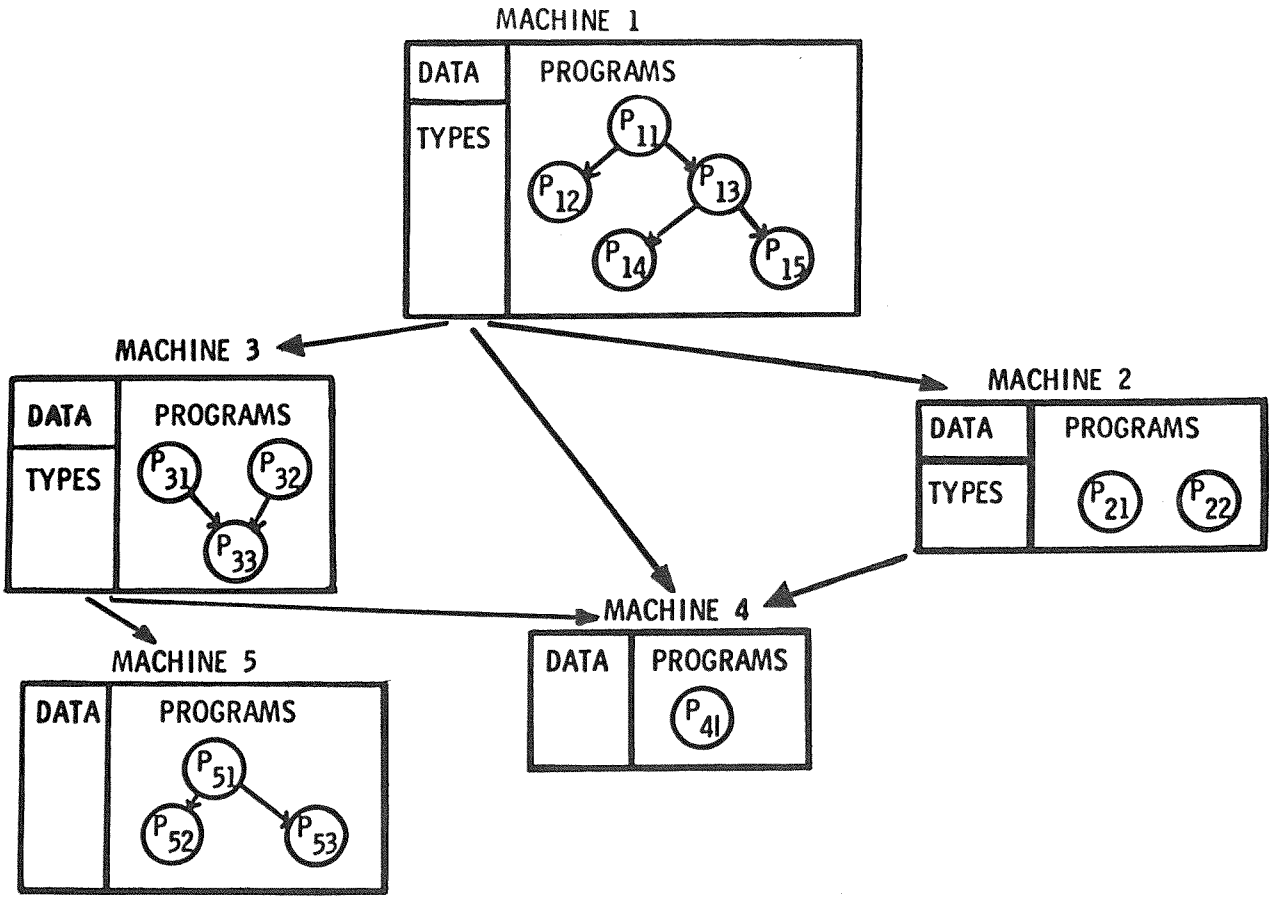
The details of the documentation scheme for a single machine are based on the principles described above. That is data, types, data invariants, and programs are to be described for each machine. However, a deeper knowledge of the theory that the one given here is required to justify the details of machine documentation. Further articles on WELLMADE are planned for publication in this magazine to cover this subject.

The use of WELLMADE is not necessarily all or nothing. While the documentation and the procedures support all the scientific principles of the diagram in Table 1 and 2, less experienced users can use WELLMADE only for its highly structured notation and documentation. More experienced users will use formal static specification techniques with or without the rigorous application of the constructive approach. Only specialized personnel, and when formal software certification is required (as it is often the case in military contracts) will use a-posteriori proof of correctness techniques.

For each level of sophistication, however, there are incremental advantages with no increase in cost (with exception of a-posteriori proofs). The graduality of use is more a matter of experience of the personnel than of development cost.
In fact, experience seems to show a reduction in cost when more sophisticated techniques, as the constructive approach, are extensively and rigorously used by experienced persons.

Fig. 1

PROGRAM AND MACHINE STRUCTURE



The benefits of WELLMADE compared to traditional techniques of program development are listed in Table 5. These benefits were estimated from the experience of the use of WELLMADE to date. Table 6 gives syntetically the situation of WELLMADE application experience.

6. CONCLUSIONS

The design methodology WELLMADE as it is today is the result of several attempts for the creation of a comprehensive methodology for the development of highly reliable software at reasonable development costs.
The use of this methodology will reduce overall development costs by drastically reducing maintenance. Although WELLMADE is based on well known existing scientific principles, it is the only known example of a methodology covering all the aspects of correct system design.

The problem of personnel education in the use of a rigorous design methodology cannot be overemphasized. In WELLMADE, the possibility to introduce different levels of formalism should be helpful for a gradual introduction of the methodology.

7. REFERENCES

[BOEH75] : B.Boehm, R.K.McClean, and D.B. Urfrig, "Some Experience with Automated Aids to the Design of Large-Scale Reliable Software", IEEE Transactions on Software Engineering, vol.SE-1, no.1 (March 1975)

[MILL76] : H.D.Mills, "Software Development", presented at the 2nd International Conference on Software Engineering, 13-15 October 1976, San Francisco, California

[FLOY67] : R.W.Floyd, "Assigning Meaning to Programs", Proceedings of the Symposium in Applied Mathematics, vol. 19, J.T.Schwartz (ed.), American Mathematical Society, Providence, RI (1967)

[HOAR69] : C.A.R.Hoare, "An Axiomatic Approach to Computer Programming", Communications of the ACM, vol.12,no.10 (October 1969)

[BOYD78] : D.L.Boyd and A.Pizzarello, "Introduction to the WELLMADE Design Methodology", 3rd International Conference on Software Engineering, May 10-12

Table 5

WELLMADE BENEFITS
COMPARED TO TRADITIONAL TECHNIQUES

STRUCTURE	- Higher productivity Improved code readability ⟶ easier maintenance
STATIC SPECS	- Much higher reliability Indispensable basis for certification Much easier maintenance
CONSTRUCTIVE METHODS	- Higher productivity for experienced persons Much higher quality Indispensable for formal certification
PROOFS	- Higher cost (need specialized personnel) Formal certification

1978, Atlanta, Georgia, and IEEE Transactions on Software Engineering, July 1978

[DIJK68] : E.W.Dijkstra, "A Constructive Approach to the Problem of Program Correctness", BIT 8 (1968)

[DIJK76] : E.W.Dijkstra, "A Discipline of Programming", Prentice-Hall, Englewood Cliffs, N.J., 1976

[PIZZ77] : A.Pizzarello, "The Constructive Approach", Honeywell LSEO Internal Report, 1977

[HOAR72] : C.A.R.Hoare, "Proof of Correctness of Data Representation", Acta Informatica, vol.1, fasc.4 (1972)

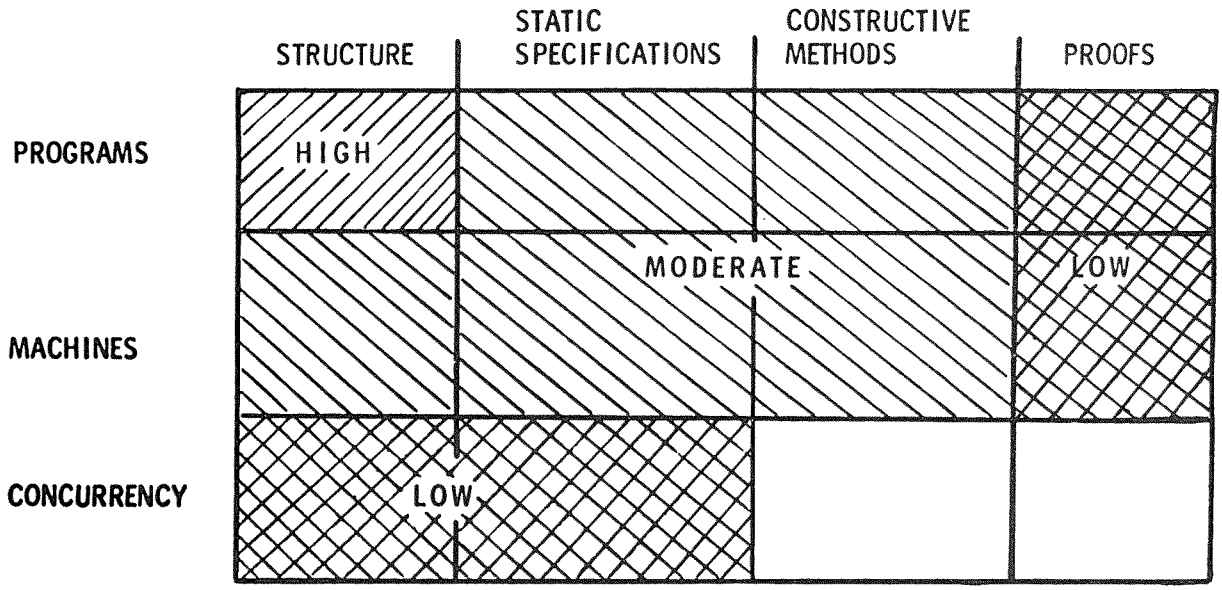
[WULF76] : W.A.Wulf, R.A.London and M.Shaw, "An Introduction to the Construction and Verification of ALPHARD Programs", IEEE Transactions on Software Engineering, vol.SE-2,no.4 (December 1976)

[ROUB77] : O.Roubine and L.Robinson, "SPECIAL Reference Manual", Technical Report CSG-45, Stanford Research Institute, Menlo Park, California 94025, jan.77

[HONE78] : Honeywell Corporate Computer Sciences Center, "WELLMADE Basic Course", 1978

Table 6

WELLMADE EDUCATION & EXPERIENCE



SECOB: un precompilatore Cobol integrato per la programmazione strutturata

S. Antocicco °
C. Barassi +
M. Maiocchi ^
P. Marighella '

1. Introduzione

L'esigenza di strumenti linguistici in fase di codifica, che corrispondano agli strumenti concettuali della programmazione strutturata della fase di disegno dei programmi, viene generalmente risolta mediante l'impiego di precompilatori: ciò vale specificamente nel caso di necessità di usare linguaggi di programmazione, come Cobol e Fortran, non orientati alla programmazione strutturata [1].

L'uso di precompilatori, in genere,

- richiede la conservazione di due liste dei programmi (sorgente e compilato intermedio), in quanto esse contengono differenti tipi di informazioni; ciò procura un aumento di carico gestionale [1];
- presenta problemi di operabilità, in quanto il programmatore ha informazioni di cross reference, diagnostici Cobol, ecc. riferentisi ad una lista che non ha scritto e che può essere notevolmente differente dal sorgente originale [1];
- tende a ridurre la qualità dei programmi nel tempo, in quanto, spesso, correzioni veloci vengono apportate sul sorgente Cobol intermedio, senza la necessaria cura di conservarne la "buona" strutturazione.

Il precompilatore SECOB si indirizza alla risoluzione di tali problemi e

- permette l'organizzazione dei programmi in modo top-down, con una definizione e ridefinizione dei dati distribuita nei vari moduli;
- mette a disposizione le principali strutture di controllo della programmazione strutturata;
- integra completamente la propria lista di compilazione con quella del compilatore Cobol della macchina ospite, in modo da costruire un unico documento di riferimento contenente tutte le informazioni relative ad un programma.

Il precompilatore SECOB è stato costruito nell'ambito di una collaborazione tra l'Istituto di Cibernetica dell'Università di Milano, il Credito Italiano e la Banca Provinciale Lombarda di Bergamo. Esso è attualmente distribuito sul mercato italiano dalla Data Management.

2. Struttura del precompilatore SECOB

Il precompilatore SECOB è costituito da due programmi distinti, che si integrano con il compilatore Cobol della macchina ospite, senza tuttavia alcun intervento di modifica su di esso (fig.1).

° Confindustria
+ Credito Italiano
^ Istit.di Cibernetica - Univ.di Milano e Honeywell ISI
' Banca Provinciale Lombarda

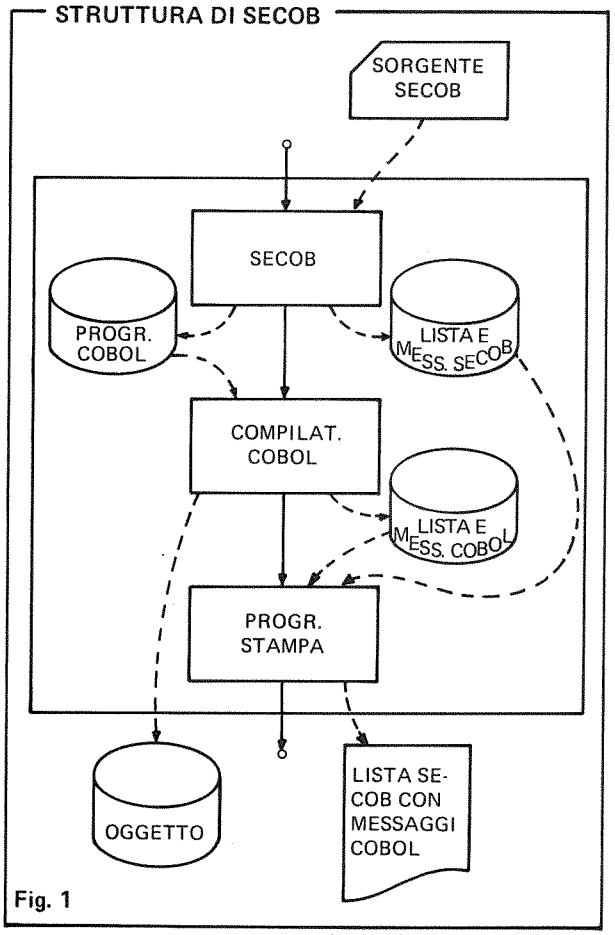


Fig. 1

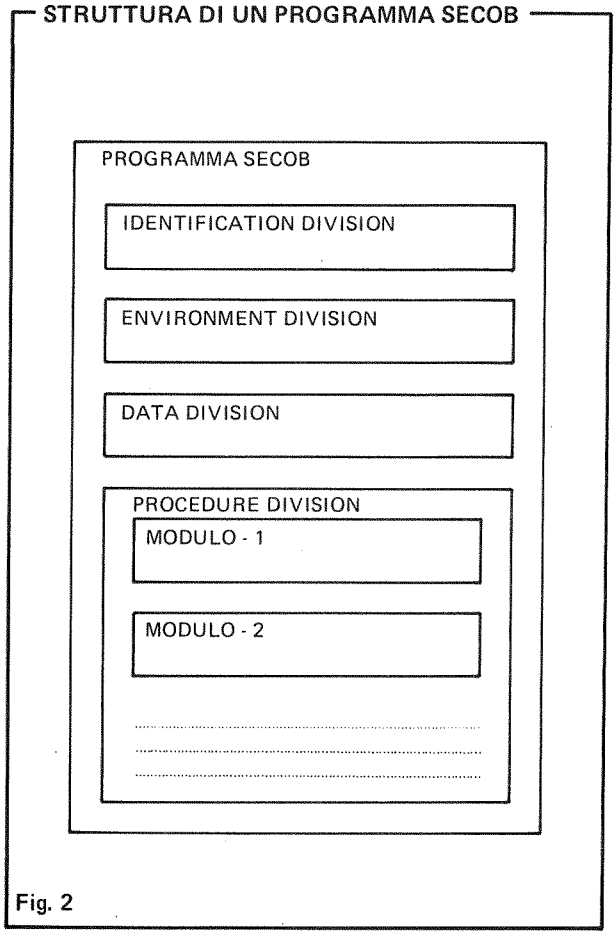


Fig. 2

In una prima fase il precompilatore vero e proprio traduce il sorgente SECOB producendo su disco sia il sorgente Cobol intermedio, sia un certo insieme di informazioni finalizzate alla costruzione del listing definitivo (immagini schede, diagnostici, informazioni sulla struttura del programma, ecc.).

Quindi il sorgente intermedio viene fornito al compilatore Cobol, che fornisce l'oggetto, ma restituisce la lista di compilazione come un'immagine della stampa, su disco.

Infine, un programma opportuno costruisce e stampa un "merge" intelligente delle due liste SECOB e Cobol; tale programma deve conoscere la struttura della lista del compilatore della macchina ospite, e ne rispetta completamente la struttura e la filosofia; esso deve ovviamente essere personalizzato per ogni tipo di compilatore impiegato.

3. Il sorgente SECOB

Il precompilatore SECOB ha lo scopo di permettere la programmazione strutturata top-down in Cobol, e si orienta quindi a mettere a disposizione facilities per l'organizzazione gerarchica dei programmi e un certo insieme di strutture di controllo. Le estensioni del linguaggio rispecchiano comunque la filosofia del Cobol e l'apprendimento del loro uso risulta praticamente immediato da parte di chi abbia già conoscenza di tale linguaggio.

Un programma SECOB (fig. 2) risulta strutturato esattamente come un programma Cobol, nelle quattro divisioni abituali; tuttavia la Data Division non è l'unico strumento per definire dati (che possono essere descritti anche nei singoli moduli di Procedure Division) e la Procedure Division risulta organizzata in moduli di diverso livello gerarchico che si richiamano tra di loro.

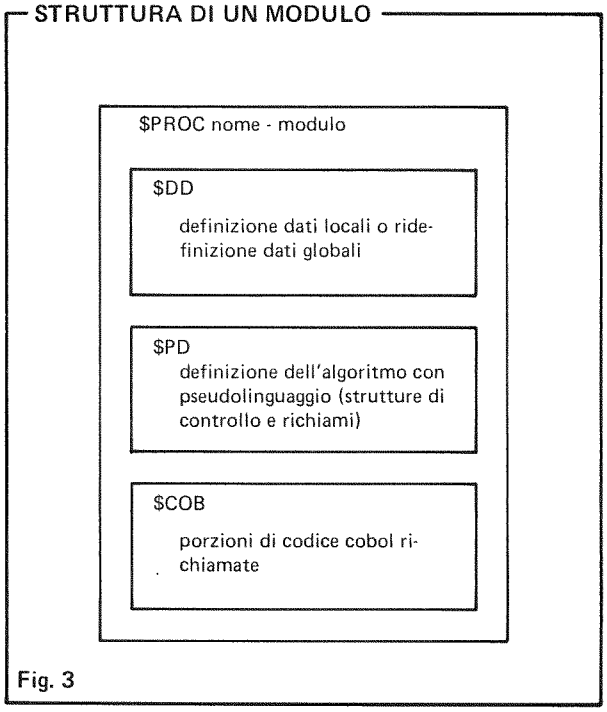


Fig. 3

Ogni modulo (fig.3) è visto come un singolo programma ed è a sua volta organizzato in tre porzioni:

- una DD (Data Division "locale"), che permette definizioni di aree di dati, secondo la solita sintassi Cobol. Tali dati sono in realtà globali a tutti gli effetti, ed il precompilatore li trasferisce nella Data Division del programma Cobol; tuttavia risulta molto utile, ai fini di un buon disegno del programma e di una sua facile lettura, che le diverse aree siano definite vicino al codice che le usa, nel momento in cui esse si rendono necessarie. Il precompilatore consente anche la ridefinizione di aree precedentemente definite, in modo tale da permettere l'implementazione di veri e propri livelli di astrazione dei dati; in tal caso la definizione è leggermente differente da quella abituale Cobol, ed ha la forma:

  nn nome-1 REDEFINES/nome-x
  nome-2

  intendendo con

  nn il numero di livello del nome ridefinito
  nome-1 il nuovo nome definito
  nome-2 il nome ridefinito
  nome-x il nome del livello 01 contenente nome-2;

- una PD (Procedure Division "locale") (fig.4), che contiene la descrizione schematica di tutto il modulo, mediante istruzioni rese disponibili dal precompilatore, che consistono esclusivamente in strutture di controllo e richiami di altri moduli o di porzioni di codice Cobol. Questa porzione non può contenere istruzioni Cobol; le strutture di controllo fornite sono l'IF.. THEN..ELSE a più vie e diversi tipi di iterazione con uscita per "vero" o per "falso" e con controllo all'inizio o alla fine del codice da iterare. I richiami riguardano tre distinti livelli: richiami di porzioni di codice Cobol inserite nello stesso modulo, richiami di altri moduli dello stesso programma o richiami di programmi compilati separatamente;

- una COB (istruzioni elementari Cobol richiamate), che contiene un insieme di porzioni di codice Cobol, ciascuna dotata di nome (come nome di paragrafo), richiamate dalla PD dello stesso modulo (o anche da PD di altri moduli dello stesso programma).

4. Conclusioni

Il precompilatore è disponibile attualmente sul calcolatore Honeywell Livello 62 e H6000/Livello 66. Esso è tuttavia trasferibile su altre macchine che accettino il Cobol ANSI, pur di costruire un programma ad hoc per la generazione del listing integrato.

Sviluppi futuri del precompilatore prevedono opzioni di ausilio al testing (funzioni di "flagger" [2]), estensioni delle strutture di controllo (iterazioni con uscite intermedie e struttura di Zahn), facilities per elaborazioni di "program inversion" [3] (allo scopo di adeguare completamente il

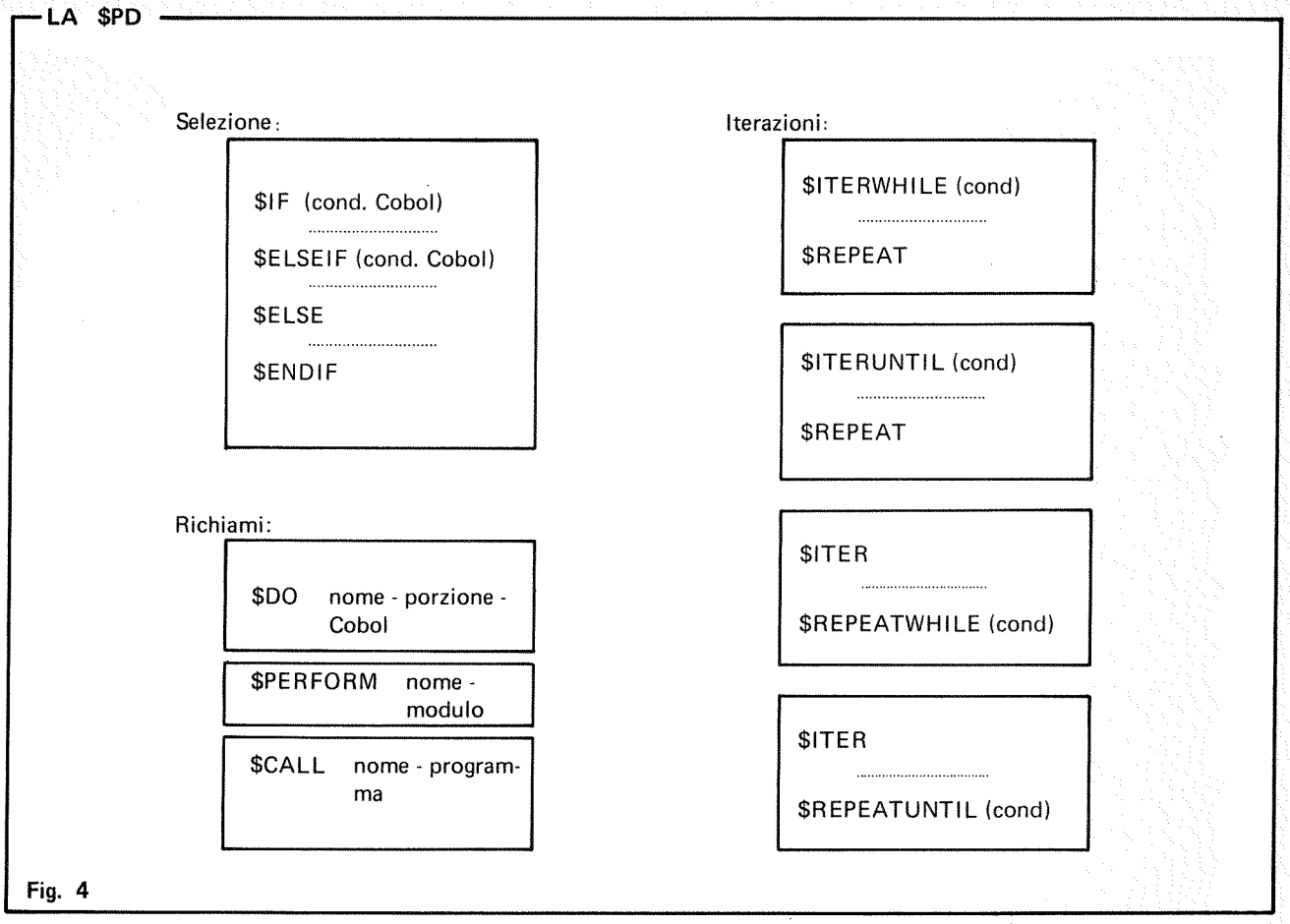


Fig. 4

SECOB a metodologie di programmazione come quella di Jackson e Phos, con le quali risulta già ora vantaggiosamente adottabile), introduzione di facilities per commentazione, elaborazione del sorgente per estrarre automaticamente documentazione. In relazione a quest'ultimo punto, il precompilatore è già in grado di produrre la stampa delle sole PD di tutti i moduli, che costituisce l'equivalente del flow-chart gerarchico dell'intero programma.

5. Riferimenti

[1] M.Maiocchi, Programmazione strutturata e precompilatori - Scuola Reiss-Romoli - Corso Sit-Siemens su Ingegneria del Software - maggio 1977

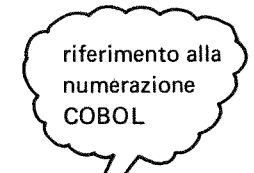
[2] F.Gariboldi, B.Giordani, G.A. Lanzarone, P.V.Renesto, Flagger: uno strumento per la valutazione dei risultati delle prove - NOTE DI SOFTWARE, n.1-gennaio-marzo 1977

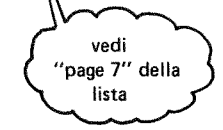
[3] M.Jackson, Principles of program design, Academic Press, 1975.

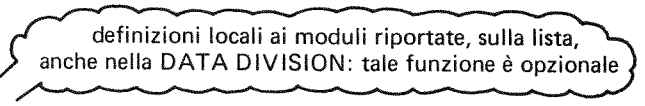
Appendice

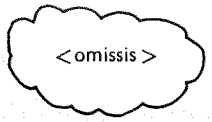
Riportiamo qui un esempio di compilazione SECOB nella versione per H6000. Per problemi di spazio, vengono presentate solo alcune parti della compilazione, che permettono tuttavia di cogliere tutti gli aspetti significativi del prodotto. Il programma compilato è una versione SECOB di quello apparso sul n.4 di NOTE DI SOFTWARE, come esemplificazione della metodologia Phos, con un archivio clienti sequenziale.

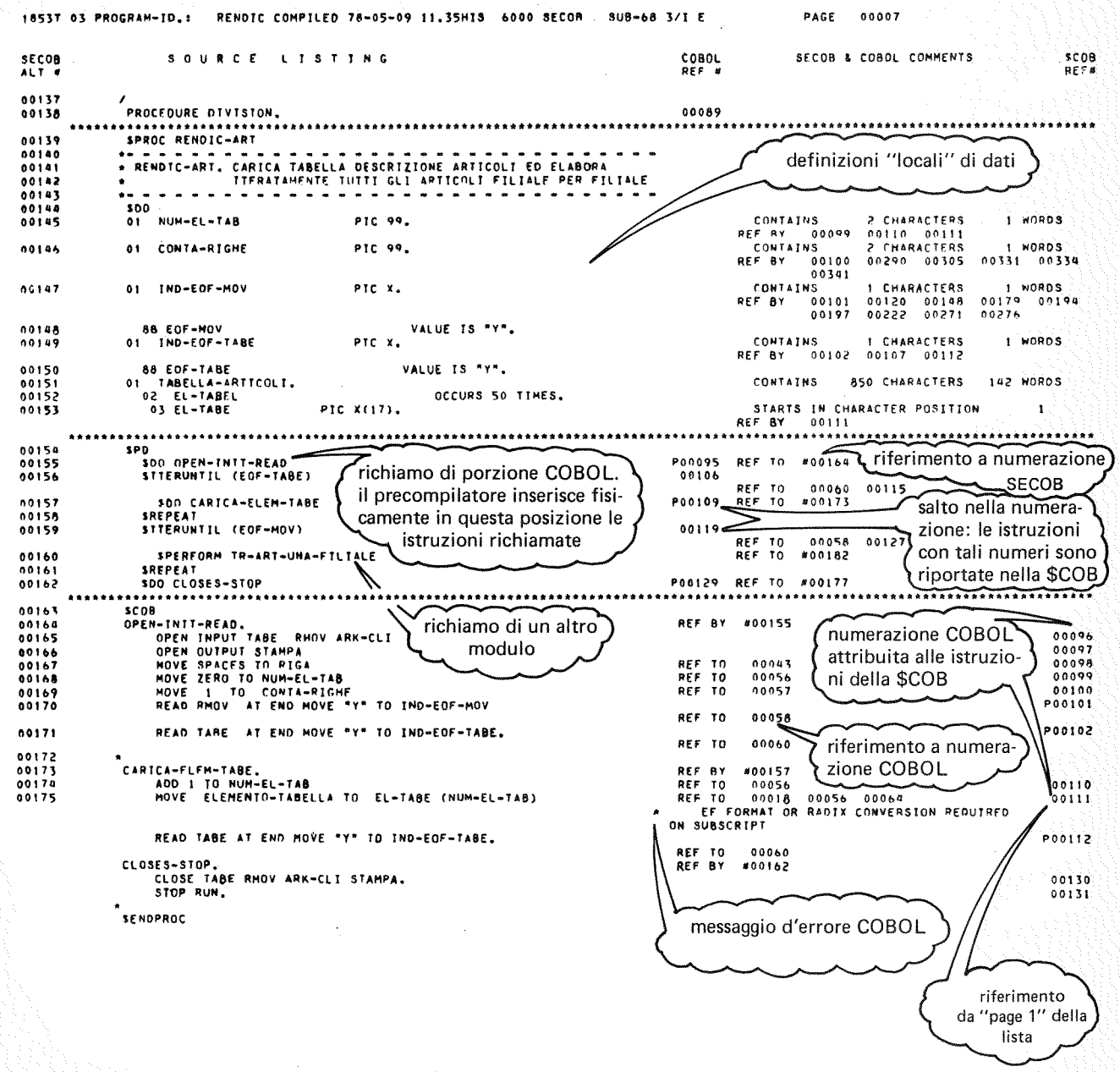
```
18537 03 PROGRAM-ID.:   RENDIC COMPILED 78-05-09 11.35HIS  6000 SECOB   SUB-68 3/I E          PAGE   00001

SECOB                          S O U R C E   L I S T I N G                          COBOL      SECOB & COBOL COMMENTS          SCOB
ALT #                                                                               REF #                              REF#

00001       IDENTIFICATION DIVISION.                                                00001
00002       PROGRAM-ID. RENDIC.                                                     00002
00003      *
00004      *- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
00005      *RENDICONTI: LEGGE  UNA  TABELLA  CON  DESCRIZIONE  E  PREZZO
00006      *
00007      *            ARTICOLI;  LEGGE  UN  FILE  DI  MOVIMENTI  DI
00008      *
00009      *            ARTICOLI  ORDINATI  PER  FILIALI/CLIENTI/CODICE
00010      *
00011      *            ARTICOLO;  STAMPA  I  TOTALI  DEI  RICAVI  PER
00012      *
00013      *            ARTICOLO  PER  FILIALE/CLIENTE  PER  TUTTI  I
00014      *
00015      *            CLIENTI  PRESENTI  SU  UN  ARCHIVIO  IDX
00016      *
00017      *- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
00018       ENVIRONMENT DIVISION.                                                  00003
00019       CONFIGURATION SECTION.                                                 00004
00020       SOURCE-COMPUTER. 6000 FIS.                                             00005
00021       OBJECT-COMPUTER. 6000 FIS.                                             00006
00022       SPECIAL-NAMES.                                                         00007
00023       INPUT-OUTPUT  SECTION.                                                 00008
00024       FILE-CONTROL.                                                          00009
00025           SELECT  TABE         ASSIGN TO A1.                                 00010
00026           SELECT  ARK-CLI      ASSIGN TO A2.                                 00011
00027           SELECT  RMOV         ASSIGN TO A3.                                 00012
00028           SELECT  STAMPA       ASSIGN TO A4 FOR LISTING.                     00013
00029      *- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
00030       DATA DIVISION.                                                         00014
00031       FILE SECTION.                                                          00015
00032      *
00033      *- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
00034      * TABE    FLUSSO SEQ. IN INPUT: CONTIENE RECORDS CON DATI RELATIVI
00035      *         AL SINGOLO ARTICOLO, DA CARICARE IN TABELLA
00036      *- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
00037      *
00038       FD  TABE  LABEL RECORD IS STANDARD                                     00016
00039                 DATA  RECORD IS  ELEMENTO-TABELLA.                          00017
00040       01  ELEMENTO-TABELLA          PIC X(17).                              00018
                                                                                  CONTAINS     17 CHARACTERS      3 WORDS
                                                                                  REF BY   00111
00041      *
00042      *
00043      *- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
00044      * ARK-CLI: FLUSSO IDX IN INPUT: CONTIENE RECORDS CON DATI RELATIVI
00045      *          AL SINGOLO CLIENTE DA USARE PER RICERCA DATI ANAGRAFICI
00046      *- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
00047      *

18537 03 PROGRAM-ID.:   RENDIC COMPILED 78-05-09 11.35HIS  6000 SECOB   SUB-68 3/I E          PAGE   00002

SECOB                          S O U R C E   L I S T I N G                          COBOL      SECOB & COBOL COMMENTS          SCOB
ALT #                                                                               REF #                              REF#

00048       FD  ARK-CLI LABEL RECORD IS STANDARD                                   00019
00049                   DATA RECORD IS REC-ARK.                                    00020
00050       01  REC-ARK                   PIC X(39).                              00021    CONTAINS      39 CHARACTERS     7 WORDS
 DEF    228 01  REC-ARK-CLI.                                                       00022    CONTAINS      40 CHARACTERS     7 WORDS
 DEF    229    02 TR-ARK   PIC X.                                                  00023    STARTS IN CHARACTER POSITION       1
 DEF    230    02 TR-ARK                  PIC X.                                  00024    STARTS IN CHARACTER POSITION       2
 DEF    231    02 COD-CLI                 PIC X(8).                               00025    STARTS IN CHARACTER POSITION       3
                                                                                  REF BY   00167
 DEF    232    02 ANAGRAFICA              PIC X(30).                              00026    STARTS IN CHARACTER POSITION      11
                                                                                  REF BY   00311  00323
00051      *
00052      *
00053      *- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
00054      * RMOV    FLUSSO SEQ.IN INPUT: ORDINATO SU FILIALE/CLIENTE/ARTICOL
00055      *         OGNI ARTICOLO HA UN RECORD DESCRIZIONE (EVENT.ASSENTE)
00056      *         E' PIU' RECORDS MOVIMENTO (EV. ASSENTI)
00057      *- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
00058      *
00059       FD  RMOV  LABEL RECORD IS STANDARD                                     00027
00060                 DATA RECORD IS REC-MOV.                                    00028
00061       01  REC-MOV                   PIC X(55).                              00029    CONTAINS      55 CHARACTERS    10 WORDS
 DEF    193 01  REC-MOV-ART.                                                       00030    CONTAINS      55 CHARACTERS    10 WORDS
 DEF    194    02  TIPO-REC               PIC 9.                                  00031    STARTS IN CHARACTER POSITION       1
                                                                                  REF BY   00218  00227
 DEF    195    02  CHIAVE.                                                        00032
 DEF    196       03  K-FILIALE           PIC X(8).                               00033    STARTS IN CHARACTER POSITION       2
                                                                                  REF BY   00142  00147
 DEF    197       03  K-CLIENTE           PIC X(8).                               00034    STARTS IN CHARACTER POSITION      10
                                                                                  REF BY   00166  00167  00178  00193
 DEF    198       03  K-ARTICOLO          PIC X(8).                               00035    STARTS IN CHARACTER POSITION      18
                                                                                  REF BY   00213  00228  00270
 DEF    199    02 DESCR                   PIC X(30).                              00036    STARTS IN CHARACTER POSITION      26
                                                                                  REF BY   00221
 DEF    337    02 DESCRIZ REDEFINES                DESCR.                          00037
 DEF    338       03 COD-MOV              PIC X(8).                               00038    STARTS IN CHARACTER POSITION      26
 DEF    339       03 QTA                  PIC 9(4).                               00039    STARTS IN CHARACTER POSITION      34
                                                                                  REF BY   00274
 DEF    340       03 FILLER               PIC X(18).                              00040    STARTS IN CHARACTER POSITION      38
00062      *
00063      *
```










definizioni "locali" di dati
richiamo di porzione COBOL. il precompilatore inserisce fisicamente in questa posizione le istruzioni richiamate
riferimento a numerazione SECOB
salto nella numerazione: le istruzioni con tali numeri sono riportate nella $COB
richiamo di un altro modulo
numerazione COBOL attribuita alle istruzioni della $COB
riferimento a numerazione COBOL
messaggio d'errore COBOL
riferimento da "page 1" della lista

```
1853T 03 PROGRAM-ID.:   RENDTC COMPILED 78-05-09 11.35HIS  6000 SECOB   SUB-68 3/I E          PAGE   00008

SECOB            S O U R C E   L I S T I N G                                  COBOL        SECOB & COBOL COMMENTS          $COB
ALT #                                                                         REF #                                        REF#
     ***********************************************************************************************************************
00182        $PROC TR-ART-UNA-FILIALE                                                  REF BY  #00160
00183       *- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
00184       * TR-ART-UNA-FILIALE: TUTTI FLI ARTICOLI DI
00185       *                     UNA FILIALE, ITERANDO CLIENTE PER CLIENTE
00186       *- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
00187       * COND.IN: TABELLA DESCR:ART. CARICATA; UN MOVIMENTO LETTO
00188       *
00189       * COND.OUT: UNA FILIALE TRATTATA; UN MOVIMENTO NUOVO LETTO
00190       *- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
00191       *
00192        $DD
00193        01  REC-MOV-ART   REDEFINES/REC-MOV.                                      CONTAINS     55 CHARACTERS    10 WORDS
00194          02  TIPO-REC              PIC 9.                                        STARTS IN CHARACTER POSITION       1
                                                                                      REF BY   00218  00227
00195          02  CHIAVE.
00196             03  K-FILIALE          PIC X(8).                                     STARTS IN CHARACTER POSITION       2
                                                                                      REF BY   00142  00147
00197             03  K-CLIENTE          PIC X(8).                                     STARTS IN CHARACTER POSITION      10
                                                                                      REF BY   00166  00167  00178  00193
00198             03  K-ARTICOLO         PIC X(8).                                     STARTS IN CHARACTER POSITION      18
                                                                                      REF BY   00213  00228  00270
00199          02 DESCR                  PIC X(30).                                    STARTS IN CHARACTER POSITION      26
                                                                                      REF BY   00221
00200       *
00201        01  DEP-COD-FIL             PIC X(8).                                     CONTAINS      8 CHARACTERS     2 WORDS
                                                                                      REF BY   00142  00147  00309
00202       *
00203        01  INDICE-CLIENTE          PIC X.                                        CONTAINS      1 CHARACTERS     1 WORDS
                                                                                      REF BY   00143  00306  00312
00204          88 NUOVO-CLIENTE          VALUE IS "1".
     ***********************************************************************************************************************
00205        $PD
00206          $DO SALVA-K-FIL                                                P00141  REF TO  #00211
00207          $ITERUNTIL (K-FILIALE NOT EQUAL DEP-COD-FIL OR EOF-MOV)         00147  REF TO   00033  00071
                                                                                      REF TO   00058  00155
00208            $PERFORM TRATTA-UN-CLIENTE                                           REF TO  #00216
00209          $REPEAT
     ***********************************************************************************************************************
00210        $COB
00211        SALVA-K-FIL.                                                      REF BY  #00206
00212            MOVE K-FILIALE TO DEP-COD-FIL.                                REF TO   00033  00071                      00142
00213            MOVE "1" TO INDICE-CLIENTE.                                   REF TO   00072                             00143
00214       *- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
00215        $ENDPROC
```

ridefinizione del livello 01 REC - MOV

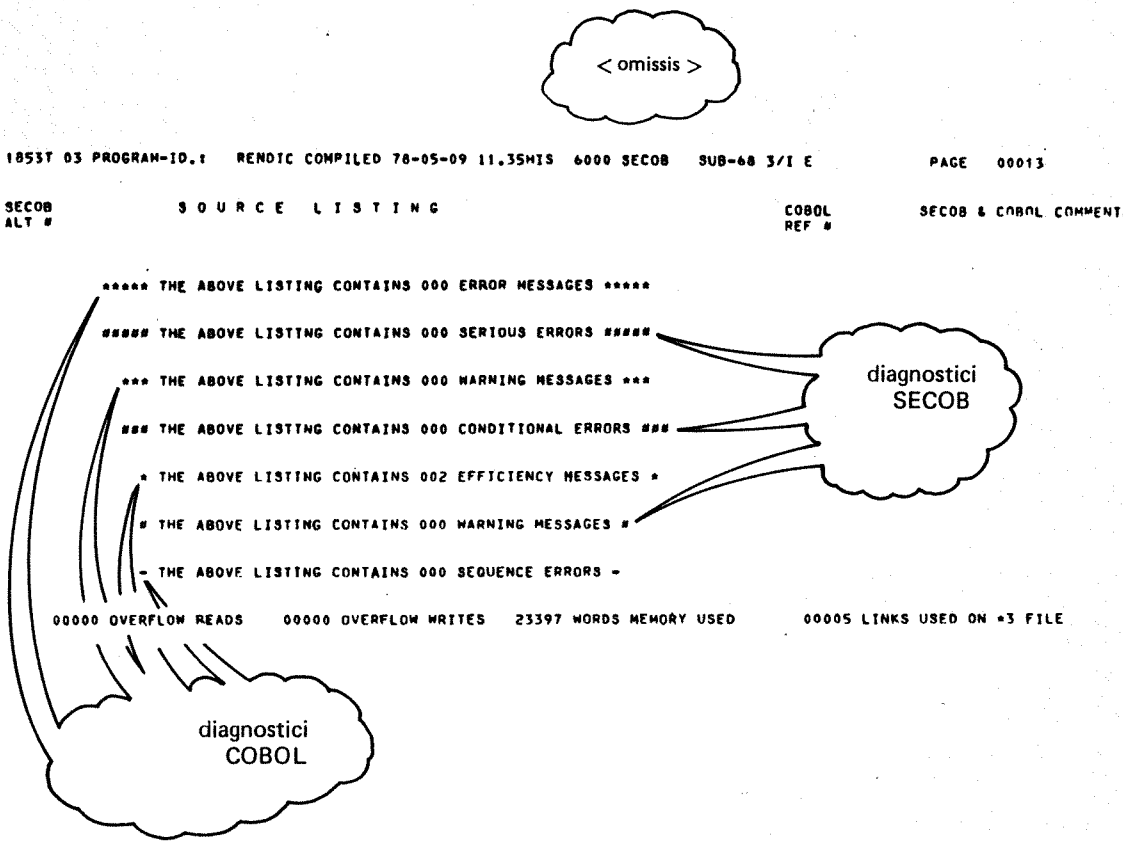
```
1853T 03 PROGRAM-ID.:   RENDIC COMPILED 78-05-09 11.35HIS  6000 SECOB   SUB-68 3/I E          PAGE   00009

SECOB              S O U R C E   L I S T I N G                                  COBOL        SECOB & COBOL COMMENTS          $COB
ALT #                                                                           REF #                                        REF#
      ***********************************************************************************************************************
00216        $PROC TRATTA-UN-CLIENTE                                                  REF BY  #00208
00217       *- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
00218       * TRATTA-UN-CLIENTE: SALTA I MOVIMENTI DEL CLIENTE SE QUESTO
00219       *    MANCA IN ARK-CLI; ALTRIMENTI TRATTA IL CLIENTE  ITERANDO
00220       *    ARTICOLO PER ARTICOLO.
00221       *- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
00222       * COND. IN: UN MOVIM. LETTO
00223       *
00224       * COND.OUT: CLIENTE TRATTATO; NUOVO MOVIMENTO  LETTO
00225       *- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
00226       *
00227        $DD
00228        01  REC-ARK-CLI   REDEFINES/REC-ARK.                                    CONTAINS     40 CHARACTERS      7 WORDS
00229          02 TR-ARK    PIC X.                                                   STARTS IN CHARACTER POSITION        1
00230          02 TR-ARK                  PIC X.                                     STARTS IN CHARACTER POSITION        2
00231          02 COD-CLI                 PIC X(8).                                  STARTS IN CHARACTER POSITION        3
                                                                                    REF BY   00167
00232          02 ANAGRAFICA              PIC X(30).                                 STARTS IN CHARACTER POSITION       11
                                                                                    REF BY   00311  00323
00233       *
00234        01  DEP-K-CLI                PIC X(8).                                  CONTAINS      8 CHARACTERS      2 WORDS
                                                                                    REF BY   00166  00178  00193  00310  00322
00235       *
00236        01 TROVATO-IN-ARK            PIC X.                                     CONTAINS      1 CHARACTERS      1 WORDS
                                                                                    REF BY   00168  00169  00174
00237          88 TROVATO                 VALUE IS "Y".
00238       *
00239        01 INDICE-ARTICOLO           PIC X.                                     CONTAINS      1 CHARACTERS      1 WORDS
                                                                                    REF BY   00170  00313  00319  00324
00240          88 NUOVO-ARTICOLO          VALUE IS "1".
      ***********************************************************************************************************************
00241        $PD
00242          $DO CERCA-CLI-IN-ARK                                          P00165  REF TO  #00253
00243          $IF (TROVATO)                                                  00173
                                                                                    REF TO   00075  00189
00244            $ITERUNTIL (K-CLIENTE NOT EQUAL DEP-K-CLI OR EOF-MOV)        00178  REF TO   00034  00074
                                                                                    REF TO   00058  00186
00245              $PERFORM TR-UN-ART                                               REF TO  #00262
00246            $REPEAT
00247          $ELSE
00248            $ITERUNTIL (K-CLIENTE NOT EQUAL DEP-K-CLI OR EOF-MOV)        00193  REF TO   00034  00074
                                                                                    REF TO   00058  00200
00249              $DO BYPASS-CLI                                            P00196  REF TO  #00259
00250            $REPEAT
00251          $ENDIF
      ***********************************************************************************************************************
00252        $COB
00253        CERCA-CLI-IN-ARK.                                                REF BY  #00242
00254            MOVE K-CLIENTE TO DEP-K-CLI                                  REF TO   00034  00074                      00166
00255            MOVE K-CLIENTE TO COD-CLI                                    REF TO   00025  00034                      00167
00256            MOVE "Y" TO TROVATO-IN-ARK                                   REF TO   00075                             00168
                 READ ARK-CLI END          MOVE "N" TO TROVATO-IN-ARK.                                                     P00169
                                                                              REF TO   00075
                 MOVE "1" TO INDICE-ARTICOLO.                                 REF TO   00077                             00170
            BYPASS-CLI.                                                       REF BY  #00249
                 READ RMOV AT END MOVE "Y" TO IND-EOF-MOV.                                                                P00197
                                                                              REF TO   00058
            $ENDPROC
```

indentazione automatica

<omissis>

```
1853T 03 PROGRAM-ID.:  RENDIC COMPILED 78-05-09 11.35MI8  6000 SECOB  SUB-68 3/I E          PAGE  00011

SECOR                    S O U R C E   L I S T I N G                                  COBOL          SECOB & COBOL COMMENTS          $COB
ALT #                                                                                 REF #                                          REF#
*****************************************************************************************************************************************
00304       $PROC STAMPA.                                                                      REF BY  #00288
00305      *- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
00306      * STAMPA
00307      *          CERCA DESCRIZIONE DEL PRODOTTO IN TABELLA; CALCOLA IL
00308      *          TOTALE DELL'IMPORTO DI TUTTI I MOVIMENTI; STAMPA L'INTE-
00309      *          STAZIONE DEL MODULO E STAMPA I TOTALI
00310      *- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
00311      * COND. IN: ARTICOLO DA STAMPARE; PRIMO RECORD MOVIMENTO LETTO
00312      *
00313      * COND.OUT:ARTICOLO TRATTATO;STAMPE EFFETTUATE;NUOVO RECORD LETTO
00314      *- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
00315      *
00316       $DD
00317      *
00318        01  RIGA-TOT REDEFINES/RIGA.                                                     CONTAINS      34 CHARACTERS      6 WORDS
                                                                                           REF BY   00288  00289
00319          02 TOT-ART-ST              PIC B(2)9(11).                                    STARTS IN CHARACTER POSITION        1
                                                                                           REF BY   00286
00320          02 IVA-ST                  PIC B(4)9(11).                                    STARTS IN CHARACTER POSITION       14
                                                                                           REF BY   00287
00321          02 FILLER                  PIC X(6).                                         STARTS IN CHARACTER POSITION       29
00322        01  IND-EL-TAB               PIC 99.                                           CONTAINS       2 CHARACTERS      1 WORDS
                                                                                           REF BY   00251  00261  00263  00274  00287
00323      *
00324        01  TOTALE                   PIC 9(11).                                        CONTAINS      11 CHARACTERS      2 WORDS
                                                                                           REF BY   00252  00275  00286  00287
00325      *
00326        01  TROVATO-IN-TAB           PIC X.                                            CONTAINS       1 CHARACTERS      1 WORDS
                                                                                           REF BY   00253  00258  00263
00327          88 PRESENTE-IN-TAB VALUE IS "Y".
00328        01  RTAB REDEFINES/TABELLA-ARTICOLI.                                           CONTAINS     850 CHARACTERS    142 WORDS
00329          03  EL-TAB-ART  OCCURS 50.
00330              04  TIPO-REC-TAB         PIC X.                                        STARTS IN CHARACTER POSITION        1
00331              04  COD-ART-TAB          PIC X(8).                                     STARTS IN CHARACTER POSITION        2
                                                                                           REF BY   00263
00332              04  PRU-TAB              PIC 9(6).                                     STARTS IN CHARACTER POSITION       10
                                                                                           REF BY   00274
00333              04 PERC-IVA-TAB         PIC 99.                                       STARTS IN CHARACTER POSITION       16
                                                                                           REF BY   00287
00334      *
00335        01  IMPONIBILE               PIC 9(10).                                        CONTAINS      10 CHARACTERS      2 WORDS
                                                                                           REF BY   00274  00275
00336      *
00337          02 DESCRIZ REDEFINES/REC-MOV     DESCR.
00338              03 COD-MOV             PIC X(8).                                     STARTS IN CHARACTER POSITION       26
00339              03 QTA                 PIC 9(4).                                     STARTS IN CHARACTER POSITION       34
                                                                                           REF BY   00274
00340              03 FILLER              PIC X(18).                                    STARTS IN CHARACTER POSITION       38
*****************************************************************************************************************************************
00341       $PD
00342         $DO INIZIAL-TOT                                                  P00250  REF TO  #00352
00343         $ITERUNTIL (PRESENTE-IN-TAB)                                      00257
                                                                                       REF TO   00086  00266
00344           $DO TR-ELEM-TAB                                                P00260  REF TO  #00356
00345         $REPEAT
00346         $ITFRUNTIL OK-ARTICOLO NOT EQUAL DEP-K-ART OR EOF-MOV)            00270  REF TO   00035  00079
                                                                                       REF TO   00058  00279
00347           $DO TR-MOV                                                     P00273  REF TO  #00361
00348         $REPEAT
00349         $PERFORM TR-INTESTAZ                                                   REF TO  #00373
00350         $DO STAMPA-TOT                                                   P00285  REF TO  #00366
*****************************************************************************************************************************************
00351       $COR
00352       INIZIAL-TOT.                                                       RFF BY  #00342
00353           MOVE ZERO TO IND-EL-TAB                                        REF TO   00084                                  00251
00354           MOVE ZERO TO TOTALE                                            REF TO   00085                                  00252
00355           MOVE "N" TO TROVATO-IN-TAB.                                    REF TO   00086                                  00253
00356       TR-ELEM-TAB.                                                       REF BY  #00344
00357           ADD 1 TO IND-EL-TAB.                                           REF TO   00084                                  00261
00358           IF DEP-K-ART = COD-ART-TAB (IND-EL-TAB)                                                                        00262
00359              MOVE "Y" TO TROVATO-IN-TAB.                                 REF TO   00068  00079  00084  00086            00263
                                                                             *     EF FORMAT OR RADIX CONVERSION REQUIRED
                                                                            ON SUBSCRIPT
00360      *
00361       TR-MOV.                                                           REF BY  #00347
00362           MULTIPLY PRU-TAB (IND-EL-TAB) BY QTA GIVING IMPONIBILE        REF TO   00039  00069  00084  00088            00274
00363           ADD IMPONIBILE TO TOTALE                                      REF TO   00085  00088                           00275
00364           READ RMOV AT END MOVE "Y" TO IND-EOF-MOV.                                                                      P00276
                                                                              REF TO   00058
00365      *
00366       STAMPA-TOT.                                                       REF BY  #00350
00367           MOVE TOTALE TO TOT-ART-ST                                     REF TO   00045  00085                           00286
00368           COMPUTE IVA-ST = (TOTALE * PERC-IVA-TAB (IND-EL-TAB)) / 100   REF TO   00046  00070  00084  00085            00287
00369           WRITE RIGA-TOT                                                REF TO   00041  00044                           00288
00370           MOVE SPACES TO RIGA-TOT                                       REF TO   00044                                  00289
00371           ADD 1 TO CONTA-RIGHE.                                         REF TO   00057                                  00290
00372       $ENDPROC
```

uso locale di istruzioni di controllo COBOL


< omissis >
1853T 03 PROGRAM-ID.: RENDIC COMPILED 78-05-09 11.35HIS 6000 SECOB SUB-68 3/I E PAGE 00013
SECOB ALT #
S O U R C E L I S T I N G
COBOL REF #
SECOB & COBOL COMMENTS
SCOB REF#
***** THE ABOVE LISTING CONTAINS 000 ERROR MESSAGES *****
THE ABOVE LISTING CONTAINS 000 SERIOUS ERRORS
*** THE ABOVE LISTING CONTAINS 000 WARNING MESSAGES ***
THE ABOVE LISTING CONTAINS 000 CONDITIONAL ERRORS
* THE ABOVE LISTING CONTAINS 002 EFFICIENCY MESSAGES *
# THE ABOVE LISTING CONTAINS 000 WARNING MESSAGES #
- THE ABOVE LISTING CONTAINS 000 SEQUENCE ERRORS -
00000 OVERFLOW READS 00000 OVERFLOW WRITES 23397 WORDS MEMORY USED 00005 LINKS USED ON *3 FILE
diagnostici SECOB
diagnostici COBOL

GENERAZIONE AUTOMATICA DI UNITA' DI PROGRAMMA: UN APPROCCIO

E. Fischetti(°), G. Petraglia(°)

1. INTRODUZIONE

Nella ricerca di nuovi strumenti che consentono di migliorare le prestazioni di un sistema informativo devono senz'altro essere prese in considerazione quelle soluzioni che tendono ad elevare il livello linguistico programmativo. Molti tentativi e realizzazioni sono stati compiuti in tale direzione ma essi in genere risultano, a nostro avviso, pienamente validi per l'impostazione metodologica scelta ma di utilizzazione limitata a specifiche applicazioni.

E' nostra opinione che dovrebbe essere definito uno strumento in grado di soddisfare le seguenti esigenze:

a) consentire al programmatore una codifica più semplice e immediata del problema

b) ridurre il numero di errori di codifica diagnosticabili solo al run-time (quali, ad esempio, quelli legati al formato dei dati e alla struttura dei files)

c) permettere l'esecuzione dei programmi e di conseguenza di intere procedure, anche se incomplete, allo scopo di poter procedere al loro controllo di validità.

In precedenti lavori (1,2) abbiamo pensato di estendere il concetto della "programmazione modulare" intendendola non come tecnica di costruzione di programmi a partire da unità elementari (moduli), bensì come un'autentica metodologia di programmazione. Questa impostazione riflette la nostra profonda convinzione che la presenza sin dallo stadio iniziale di definizione di una procedura di uno strumento avente i requisiti suddetti può permettere di caratterizzare e di costruire tutta una serie di componenti, a livello programmi, che, una volta messi insieme, consentano innanzitutto di controllare la correttezza semantica di una procedura e quindi di catalogarla e gestirla.

In questa ottica risulta evidente che lo strumento da noi proposto non risulta da solo sufficiente ma richiede l'utilizzazione di altre "facilities" ben note (ad esempio, command languages, JCL in ambito L62, utility programs, etc.).

2. SCHELETRI DI PROGRAMMA

La suddetta estensione del concetto di modularità fa sì che esso diventi uno strumento operativo del sistemista più che del programmatore. Perchè ciò sia possibile è necessario che si guardi agli archivi come gli elementi base di una procedura e alle loro interazione come "macro-moduli" della stessa. A questo livello un macro-modulo non è un programma operativo ma un insieme di programmi che gestiscono l'algoritmo di trasformazione delle informazioni da alcuni archivi verso altri (3). Un'ulteriore caratterizzazione si viene ad avere allorchè questo macro-modulo, decomposto in unità di programma, viene ad inserirsi nell'ambito di un JCL: in tale momento gli archivi vengono caratterizzati in formato e contenuto delle informazioni e le funzioni di ciascuna unità di programma diventano univocamente determinate.

Perchè questo approccio possa essere realizzato con il computer sin dalla fase iniziale abbiamo pensato di utilizzare, insieme ad altre facilities, un processor costruttore di "scheletri di programma" (denotato nel seguito come "PSP"). Tali scheletri possiedono una parte univocamente definita che riguar-

(°) Istituto di Scienze dell'Informazione - Università di Salerno

da le caratteristiche degli archivi da essi gestiti e le asserzioni del programma e una parte dummy nella quale verrà successivamente inserito l'algoritmo che è di competenza del programmatore. Questa carenza non è essenziale nell'ambito della costruzione di un progetto EDP in quanto si è comunque reso possibile, prima ancora di codificare i programmi, effettuare il test sulla gestione di una procedura meccanografica.

Per quanto riguarda i vantaggi di un tale approccio, ci limitiamo a notare che, quando la parte dummy viene codificata, il programma viene collaudato nell'ambito di tutto un contesto operativo già testato. Inoltre, la gestione di una procedura meccanografica può comunque essere iniziate anche se risultano ancora dummy alcuni moduli-programma non centrali.

Questo approccio capovolge il modo tradizionale di impostare il management di un progetto, anche se visto in termini di Chief Programmer Team (CPT). Se, ad esempio, prendiamo in considerazione gli stadi sul management previsti da Kuehne (4), anche se non sono quelli puri del CPT, vediamo che la definizione di una procedura si attua in sei stadi che vanno dallo studio di fattibilità fino alla realizzazione operativa della stessa (fig.1). Di queste fasi alcune (feasibility study, system design) vengono sviluppate mediante interazione tra l'User Department, il Chief Programmer e il Team, mentre le altre quattro vengono seguite dal sistemista interagendo con tutte le funzioni interne al CPT e con quelle esterne del CED.

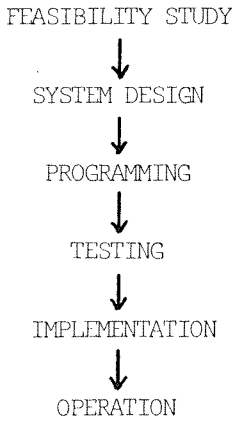


Fig. 1

Una ristrutturazione del modo di operare nell'ambito delle prime due fasi riduce enormemente il numero dei gradi di libertà che si hanno nella fase finale dovuti a una non esatta definizione degli ultimi steps sia nei tempi sia nel contenuto. Il poter non solo anticipare alcuni steps ma completare un disegno operativo già nella seconda fase dà migliori garanzie globali circa la gestione delle ultime fasi.

Esaminiamo in dettaglio quali sono gli steps su cui intervenire. In fig.2 sono riportati alcuni steps centrali della fase "feasibility" e possiamo vedere le differenze esistenti tra un sistema classico e quello che nasce a seguito della suddetta impostazione. In particolare, la definizione tradizionale dell'input e dell'output viene integrata con la definizione di tutti gli archivi necessari a gestire la procedura. Nello sviluppo di questo step il sistemista utilizza il calcolatore per una caratterizzazione formattata di tutti gli archivi servendosi di un file appropriato (nel seguito "FILE"). Questa tecnica, oltre ai vantaggi che si vedranno successivamente, permette la costruzione di una documentazione automatica degli archivi leggibile mediante terminale da programmatori ed eventuali utenti.

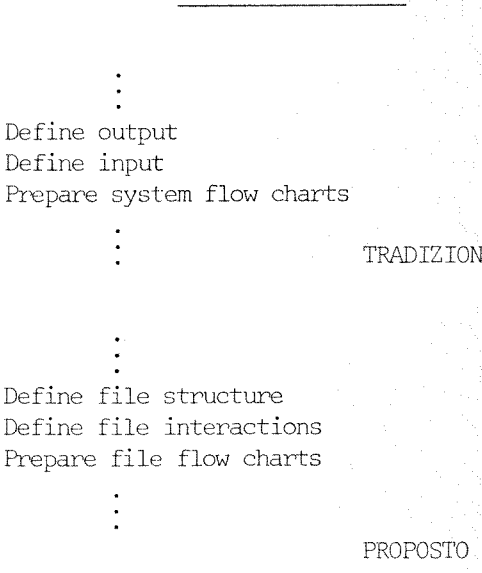


Fig. 2

Il secondo stadio (fig.3) subisce le più profonde modifiche sia sul modo di formattare le informazioni relative agli archivi sia sulla tecnica di costruzione del flow chart. L'usuale tecnica di documentazione dei files viene completamente automatizzata dal programmatore mediante il lancio di un opportuno step che aggiorna il suddetto FILE. Inoltre, il flow chart dettagliato di definizione della procedura diventa un JCL-flow chart, cioè un flow che caratterizza i moduli-programma nell'ambito di un JCL.

SYSTEM DESIGN

⋮

Prepare preliminary system flow charts
Prepare report layouts
Prepare input data elements
Approve report layouts
Approve source documents
Prepare record layouts and file organization
Complete detailed system flow charts

⋮

TRADIZIONALE

⋮

Prepare preliminary JCL flow charts
Prepare FILE
Approve FILE components
Complete detailed JCL flow charts
Generation of program skeletons with narrative
Testing of the skeletons of the procedure

⋮

PROPOSTO

Fig. 3

La validità di questa impostazione sta nella possibilità di eseguire con il computer un tale JCL-flow chart e ciò è possibile se possiamo disporre di una libreria di moduli-programma: esso, stanti le caratteristiche standard del FILE, può servirsene grazie ad opportune direttive fornite al processor. Non entriamo nel dettaglio delle altre fasi in quanto sono intuibili le modifiche da apportare.

3. STRUTTURA DEL PROCESSOR

Abbiamo costruito, in COBOL, un prototipo di processor (fig.4) che genera scheletri di programma in COBOL. Nel disegnare il processor non abbiamo utilizzato il macro-processor COBRA disponibile sull'L62 sia perchè il nostro obiettivo non era quello di generare delle routines standardizzabili, sia, e soprattutto, in quanto le parti standard possono essere gestite direttamente da PSP eliminando in tal modo un'ulteriore fase di precompilazione. Abbiamo pertanto utilizzato una tecnica del tutto simile ai linguaggi parametrici che consente di esplodere le specifiche in istruzioni COBOL. Ovviamente tale impostazione non esclude la possibilità di utilizzare COBRA nella fase di implementazione degli scheletri, allorchè PSP ha svolto interamente le sue funzioni.

Come è noto, il linguaggio COBOL è caratterizzato da divisioni di cui alcune di tipo essenzialmente dichiarativo (Identification, Environment, Data). La generazione della Environment e della Data (File Section) deve tener conto dei requisiti esposti nell'introduzione e pertanto richiede la presenza del FILE richiamabile da PSP secondo le specifiche trasmessegli dal programmatore.
Nell'approntare il prototipo abbiamo notato che queste ultime riguardano essenzialmente il nome del file, il tipo di accesso e il tipo di open: altre specifiche sono ridondanti. Ad esempio, l'utilizzo del Filler in sostituzione della definizione dettagliata del record nell'ambito della File Section associata a un certo file è solo una comodità del programmatore per ridurre il volume della codifica più che un'esigenza specifica del programma. Inoltre, la presenza di alcune dichiarative non gestite da un programmatore non costituisce affatto un inconveniente ma migliora bensì l'implementazione degli scheletri.

Ad un esame non approfondito sembrerebbe che sia avvenuto il trasferimento della codifica della File Section ad una fase più a monte e ciò in quanto deve necessariamente esistere un momento di creazione del FILE. Una più attenta considerazione delle cose mostra però le migliori prestazioni connesse a un tal modo di operare. Innanzitutto dobbiamo notare che poichè ogni procedura è costituita in media da 40-50 programmi e da 5-6 archivi abbiamo una media di dieci programmi che utilizzano lo stesso archivio. Inoltre, poichè

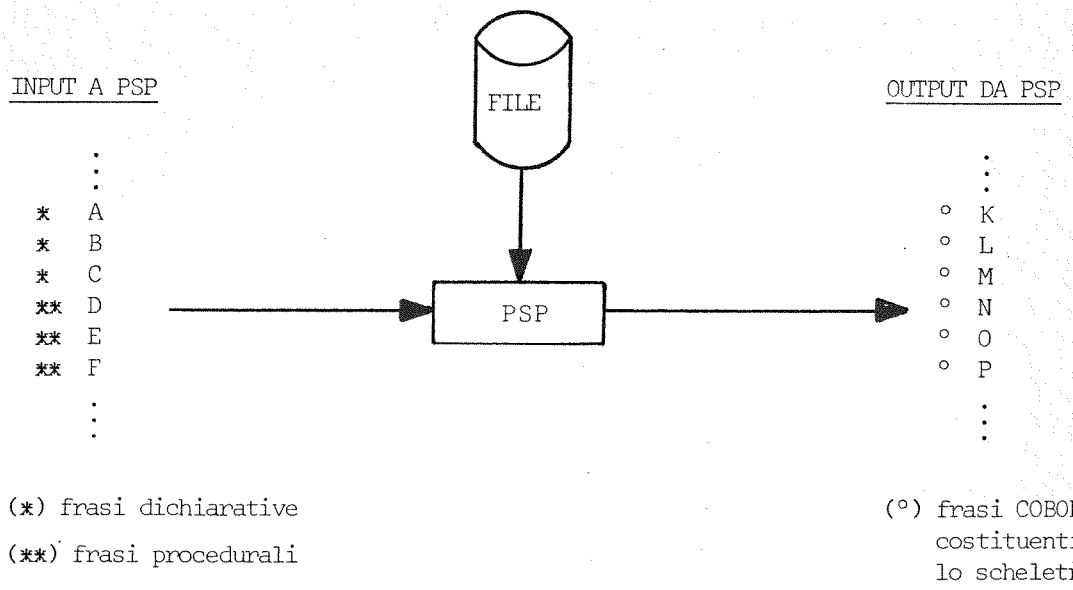


Fig. 4

il programma viene costruito utilizzando questo FILE, ottenuto in modo standard, un'eventuale modifica degli archivi della procedura richiede soltanto, anche in un contesto operativo, allorchè i programmi non sono più in forma di scheletro, l'utilizzo di tale FILE modificato per caratterizzare nuovamente la File Section, senza dover intervenire direttamente sui programmi.

Anche qui si impone il paragone con il macro-processor: osserviamo soltanto che non è possibile utilizzare quest'ultimo in quanto esso dovrebbe poter gestire intersezioni di codifiche esplose, cosa che non è consigliabile in COBRA al fine di evitare l'intersezione di routines standardizzabili.

Da quanto sinora esposto si evince che PSP, sulla base delle specifiche date, utilizza il FILE per generare sia parte delle divisioni Environment e Data sia la Job Linkage Section indispensabile per trasferire informazioni specializzate al JCL. Questo è un altro punto saliente del nostro approccio: sulla base di quanto sopra detto, al momento in cui vengono costruiti gli scheletri deve essere già catalogata la procedura perchè essa sia immediatamente testabile. E' evidente che una tale impostazione prevede una stretta interazione tra programma e JCL la qual cosa non è possibile se la Job Linkage Section viene costruita direttamente dal programmatore il quale non possiede la visione globale del progetto. A possederla è invece il sistemista al quale è devoluto il compito di costruire il FILE. Il processor, utilizzando in modo implicito una delle specifiche fornite dal programmatore (nome del programma), trasferisce, prelevandola dal FILE, l'intera Job Linkage Section nell'interno dello scheletro.

Le caratteristiche di PSP connesse alla generazione delle prime tre divisioni dello scheletro lo rendono sostanzialmente simile a un qualsiasi processor di tipo parametrico. Per la generazione della divisione Procedure si è dovuto pensare all'introduzione di un vero linguaggio che soddisfacesse i requisiti, esposti nell'introduzione, di modularità e di scheletrizzazione anche del programma.

Per meglio comprendere ciò vediamo che anche nell'ambito di una divisione Procedure è possibile una codifica top-down dei programmi sfruttando le possibilità offerte dai due verbi "Perform" e "Go to ... depending on ...". Di conseguenza una divisione Procedure può essere pensata costituita da un "monitor", immediatamente costruibile, il quale mantiene sempre il controllo dell'esecuzione del programma e da una serie di Section richiamate dal monitor che svolgono funzioni specifiche. In altri termini, PSP genera uno scheletro fornito solo di monitor e delle relative Section di tipo dummy: tali Section sono successivamente codificate dal programmatore.

La generazione automatica della parte procedurale può, in tale ottica, essere affidata a un mini-linguaggio sintatticamente ben definito le cui frasi sono fornite al processor. Tale linguaggio, la cui versione prototipo comprende i verbi Start, Move e Exit, consente, mediante opportuna traduzione in COBOL delle frasi del linguaggio, sia di costruire la successione delle varie sezioni dello scheletro, pilotate se necessario da opportuni flags, sia di caratterizzare (in modo dummy) le varie sezioni. Queste ultime sono costituite da verbi "Display" che permettono di seguire i vari passi della procedura ed eventualmente da "Accept" nel caso in cui una certa sezione lo richieda per posizionare dei flags utilizzati successivamente dal monitor per pilotare l'esecuzione di altre sezioni presenti nel programma.

4. FORMATO DELLE DIRETTIVE

L'input a PSP ha il seguente formato:

```
$   sigla   parametri
```

dove "sigla" denota la particolare funzione richiesta.

Nel prototipo realizzato le direttive ED e EDCOS realizzano la parte dichiarativa dello scheletro e individuano nel FILE il blocco di informazioni da trasferire successivamente nello scheletro. La direttiva EDIOSFCP effettua il trasferimento FILE → scheletro dell'archivio desiderato sia a livello File Control sia File Section e genera successivamente le frasi COBOL "Open" e "Close". La DDWSSGEP genera nella Working Storage Section i flags necessari per il monitor. La PDMS svolge le seguenti funzioni: terminare la parte dichiarativa dello scheletro effettuandone altresì il controllo di validità semantica, effettuare il trasferimento FILE → scheletro della Job Linkage Section interessata e infine attivare il mini-linguaggio presente in PSP. La direttiva PDSTAT denota frasi del linguaggio procedurale che vengono traslate da PSP in frasi COBOL. Infine, EN effettua i controlli finali previsti. La ridondanza delle sigle è intenzionale in quanto si prevede di implementare il prototipo attivando altre funzioni.

5. UN ESEMPIO

Allo scopo di chiarire il funzionamento di PSP in fig.5 riportiamo lo schema modulare di un programma-esempio e in fig.6 il relativo input a PSP.

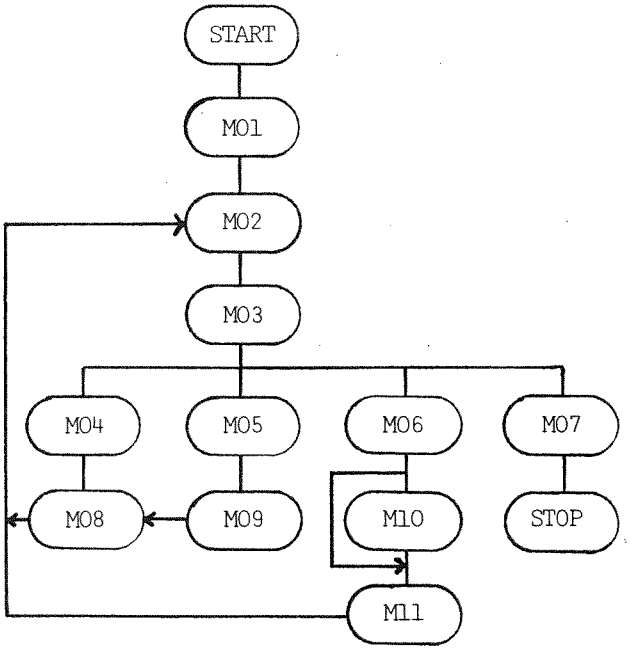


Fig. 5

```
$ID       INTERP
$EDCOS    LEVEL-62  LEVEL-62
$EDIOSFCPPRINTER    OUTPUT
$EDIOSFCPREL-FILE   I-O        DYNAMIC
$EDIOSFCPCARD-FILE INPUT
$DDWSSGEP          2
$PDMS
$PDSTAT   START M01.
$PDSTAT   MOVE M01 TO M03 THRU M02.
$PDSTAT   MOVE M03 TO M04 M05 M06 M07
          POINTER 01 SET BY M02.
$PDSTAT   MOVE M04 TO M02 THRU M08.
$PDSTAT   MOVE M05 TO M08 THRU M09.
$PDSTAT   MOVE M06 TO M10 M11 POINTER 02
          SET BY M06.
$PDSTAT   MOVE M10 TO M02 THRU M11.
$PDSTAT   EXIT M07.
$EN
```

Fig. 6

Osserviamo subito il modo in cui viene tradotto con le direttive PDSTAT lo schema stesso. In particolare, non è necessario iniziare, come nell'esempio, con "START" e terminare con "EXIT" in quanto questi due verbi sono essenzialmente dichiarativi per il mini-linguaggio e servono solo a definire il top e il bottom di uno stack necessario per la traduzione.

Il verbo "MOVE ... TO ... THRU ..." serve per generare processi sequenziali mentre il "MOVE ... TO ... POINTER ... SET BY ..." viene invece utilizzato per generare processi alternativi pilotabili grazie all'utilizzo di un flag.

Perchè la traduzione schema → frasi possa avvenire correttamente si richiede che la struttura dinamica dello schema riflessa nelle frasi sia tale da far sì che l'intero schema sia rappresentato in tutti i suoi possibili percorsi evitando però loro sovrapposizioni. Ad esempio, sarebbe scorretta la presenza di queste due frasi:

a) MOVE M01 TO M03 THRU M02.

b) MOVE M09 TO M03 THRU M08 M02.

La caratterizzazione di tutti i percorsi segue la problematica delle reti di Petri (5) in cui i moduli sono "events" relativi allo spostamento dei "markers".

Le fig.7-10 mostrano parte dello scheletro generato da PSP. Particolare interesse riveste la divisione Procedure. Infatti, se, a partire dalla fig.10, costruiamo uno schema del monitor (fig.11), vediamo che esso è equivalente all'originale anche se più esploso in conseguenza della trasformazione operata sull'input da parte dell'algoritmo presente in PSP.

```
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT PRINTER ASSIGN TO PRINT-PRU001.
    SELECT REL-FILE ASSIGN TO ANA-MSU110
    ORGANIZATION IS INDEXED
    RESERVE 2 AREAS
    RECORD KEY IS AN-MATR
    FILE STATUS IS AN-STA
    ACCESS MODE IS DYNAMIC.
    SELECT CARD-FILE ASSIGN TO CARD-TCU001.
```

Fig. 7

```
DATA DIVISION.
FILE SECTION.
FD  PRINTER
    LABEL RECORDS ARE OMITTED
    DATA RECORD IS PRINT-REC
    RECORD CONTAINS 120 CHARACTERS.
01  PRINT-REC          PIC X(120).
FD  REL-FILE
    LABEL RECORDS ARE STANDARD
    BLOCK CONTAINS 48 RECORDS
    RECORD CONTAINS 148 CHARACTERS
    DATA RECORD IS AN-SA.
*   THIS RECORD CONTAINS ALL INFORMATIONS
*   ABOUT THE STUDENT
01  AN-SA.
    02 FILLER          PIC X.
*   CODE NUMBER
    02 AN-MATR         PIC X(7).
*   NAME AND SURNAME OF THE STUDENT
*   (AN ASTERISK LOCATES THE END OF THE NAME)
    02 AN-CONO         PIC X(35).
*   BIRTH DATE (YYMMDD)
    02 AN-DANA         PIC X(2) OCCURS 3 TIMES.
*   BIRTH PLACE
    02 AN-LONA         PIC X(23).
    02 FILLER          PIC X(76).
FD  CARD-FILE
    LABEL RECORDS OMITTED
    DATA RECORD IS CARD
    RECORD CONTAINS 80 CHARACTERS.
01  CARD.
    02 CR-CODE         PIC 99.
    02 CR-KEY          PIC X(30).
    02 CR-DESCRIPT     PIC X(48).
```

Fig. 8

```
S-T-A-R-T SECTION.
A01.
    DISPLAY "INIZIO ESECUZIONE " PROG-ID.
    OPEN OUTPUT PRINTER.
    OPEN I-O REL-FILE.
    OPEN INPUT CARD-FILE.

M06 SECTION.
MM06.
    DISPLAY "SONO IN M06".
    DISPLAY "DAMMI FLAG-02".
    ACCEPT FLAG-02.
```

Fig. 9

```
 PROCEDURE DIVISION.
*       MONITOR DELLO SCHELETRO
 A00.  PERFORM S-T-A-R-T.
 A01.  PERFORM M01.
 A02.  PERFORM M02.
 A03.  PERFORM M03.
 C04.  GO TO A04 A05 A06 A07
       DEPENDING ON FLAG-01.
       DISPLAY "ERROR FLAG-01 " FLAG-01
       STOP RUN.
 A04.  PERFORM M04.
 A08.  PERFORM M08.
 A09.  PERFORM M02.
       GO TO A03.
 A05.  PERFORM M05.
 A10.  PERFORM M09.
 A11.  PERFORM M08.
       GO TO A09.
 A06.  PERFORM M06.
 C12.  GO TO A12 A13 DEPENDING ON FLAG-02.
       DISPLAY "ERROR FLAG-02 " FLAG-02
       STOP RUN.
 A12.  PERFORM M10.
 A14.  PERFORM M11.
 A15.  PERFORM M02.
       GO TO A03.
 A13.  PERFORM M11.
       GO TO A15.
 A07.  PERFORM M07.
 A16.  PERFORM S-T-0-P.
```

Fig. 10

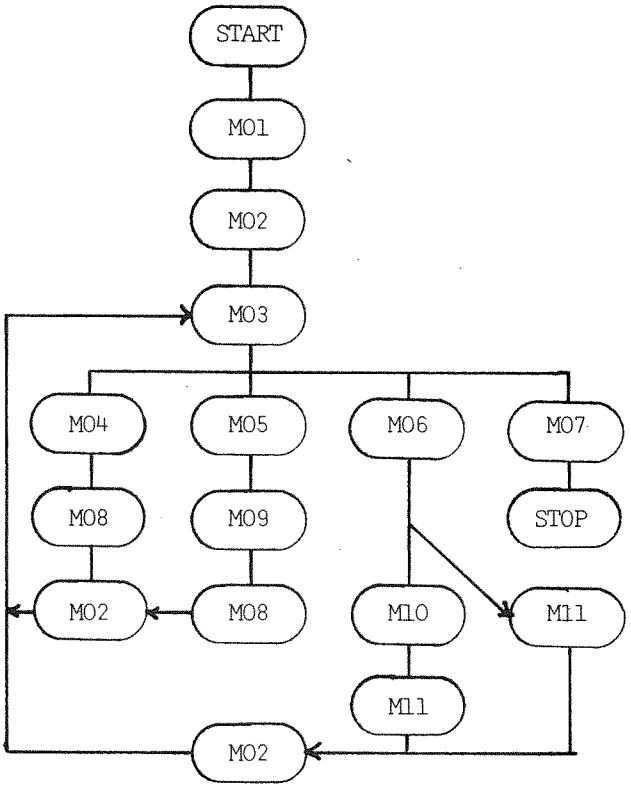


Fig. 11

6. CONCLUSIONI

L'approccio sopra delineato consente, a nostro avviso, di individuare degli strumenti di standardizzazione, basati sulla tecnica di sviluppo top-down, automatizzabili fin dalla fase di definizione di una procedura.

In particolare, è possibile pensare a dei sofisticati strumenti di preprocessing nel caso in cui quest'ultimo possa operare nell'ambito di un sistema operativo fornito di un adeguato command language. Tali preprocessors, trasferendo a monte della codifica dei programmi la definizione e il controllo della correttezza di una procedura, permettono di migliorare l'affidabilità della procedura stessa.

Abbiamo visto in PSP un valido strumento di preprocessing in quanto consente non solo la netta separazione tra il momento di costruzione dell'algoritmo e quello di caratterizzazione degli archivi ma anche di strutturare modularmente, in modo automatico, dei programmi in ambiente COBOL.

Riteniamo comunque che alcune caratteristiche dei suddetti preprocessors dovranno in futuro essere trasferite nei compilatori se si vuole ridurre il numero di steps necessari per lo sviluppo di un programma e aggiungere una nuova dimensione all'attività programmativa.

7. RIFERIMENTI

1. Fischetti,E. and Petraglia,G.:"Data base management",Atti 4° CNCB,Siena,p.179,1976.
2. Fischetti,E. e Petraglia,G.:"Scheletri di programma",Atti "AICA 77",Pisa,p.169,1977.
3. Galdi,G. e Petraglia,G.:"Progettazione modulare",Atti "AICA 77", Pisa, p.171, 1977.
4. Kuehne et al.:"Manual of computer documentation standards",Prentice-Hall, 1972.
5. Winkowski,J.:"An algebraic characterization" Inform.Proc.Letters,6, 4, p.105, 1977.

LA GENERAZIONE AUTOMATICA DEI DATI DI TEST

G. Ferraro (°), D. Marini (+)

1. Introduzione

Tra i complessi problemi che si incontrano nel testing dei programmi, come la gestione delle prove, la pianificazione dell'attività, la scelta della strategia ottimale, ve n'è uno di particolare rilevanza. Si tratta del problema della scelta dei dati di prova, che, nel caso di programmi di notevoli dimensioni, non può certo essere risolto in modo approssimativo o basandosi sull'intuizione, ma deve trovare una soluzione che si avvalga il più possibile di strumenti generali e automatici.

I vari metodi ed approcci alla scelta dei dati di test si possono considerare distribuiti in uno spettro di "filosofie". Un estremo è rappresentato dal 'test funzionale', una filosofia che pone l'accento sulla verifica delle funzionalità del programma trascurando in larga misura gli aspetti legati al codice che le realizza, e considerando quindi il programma come una 'black box' che deve fornire certe uscite a fronte di certi ingressi. L'altro estremo è rappresentato dal 'test del codice', in cui, trascurando le specifiche funzionali, si cerca di provare ogni elemento del programma studiandone la struttura interna (°°).

Le metodologie che hanno avuto maggior sviluppo si collocano per lo più nel secondo estremo dello spettro; esse cercano di correggerne gli inconvenienti mettendo in relazione le classi di dati di test individuate, con i differenti casi del problema. Tali approcci basano la scelta dei dati di prova sui possibili percorsi del controllo nella struttura del programma. Nel caso di programmi strutturati, tali percorsi, che nel seguito chiameremo cammini di controllo, corrispondono alle possibili differenti computazioni, e ai differenti casi del problema per la cui soluzione il programma è stato escogitato (cfr. 2).
Una volta noti i possibili cammini di controllo di un programma, organizzabili in classi di equivalenza (i cammini di una data classe differiscono tra loro solo per il numero di volte in cui un ciclo di iterazione è percorso), il passo successivo consiste nella ricerca di singoli valori, o di insiemi di valori di input per il programma, che provochino l'esecuzione di particolari cammini appartenenti alle varie classi di equivalenza. Sarà quindi possibile eseguire un insieme di prove con dati appartenenti alle varie classi così individuate, e di conseguenza esercitare il programma per tutti i differenti possibili casi di computazione. In tal modo, si avrà la garanzia di aver provato non solo ogni istruzione, ma anche ogni ramificazione del programma.

La metodologia di scelta dei dati di prova basata sulla identificazione dei cammini di controllo di un programma richiede una tecnica che consenta, noto un cammino di controllo particolare, di ricavare l'insieme dei dati di input che ne provochino l'esecuzione. Un tale insieme può essere adeguatamente descritto da una famiglia di proprietà o relazioni cui le variabili di input del programma dovranno soddisfare. Chiamiamo tale famiglia predicato di test (3). Un predicato di test può essere inteso come la congiunzione di tutte le condizioni che devono essere verificate da un dato di input affinchè un determinato cammino venga percorso. Tali condizioni non sono altro che le condizioni presenti in tutti i

(°) G.F. Termotecnica - Legnano

(+) Università degli Studi di Milano - Istituto di Cibernetica

(°°) Un interessante lavoro contenente anche elementi di rassegna sugli approcci alla scelta dei dati di test è (1).

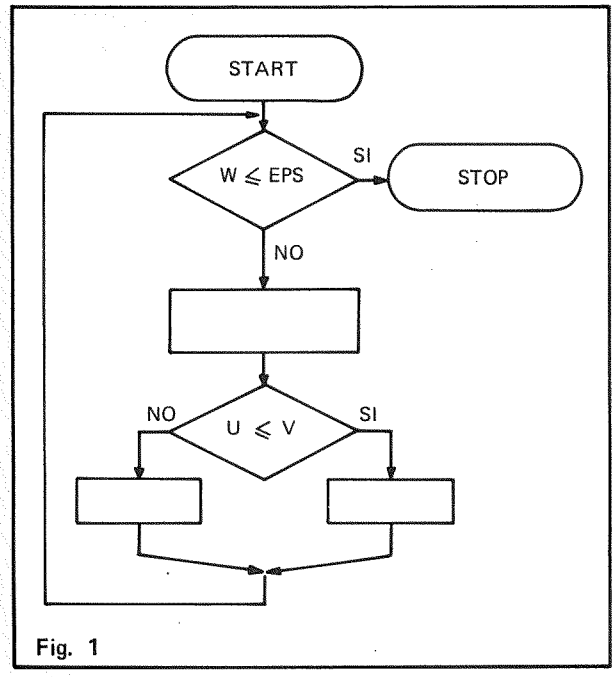


Fig. 1

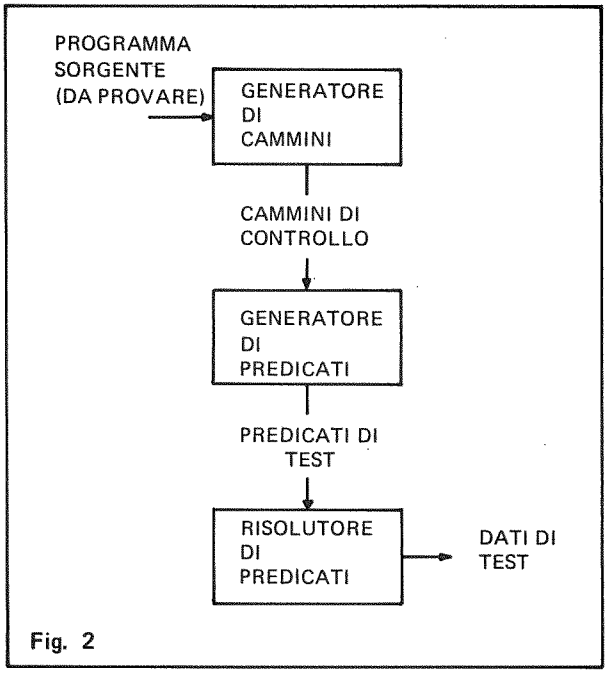


Fig. 2

test di controllo che si incontrano lungo il cammino in questione. Ad esempio, consideriamo il programma rappresentato in fig. 1. Un predicato di test, corrispondente a un cammino di controllo, è rappresentabile come $W > EPS$ and $U \leq V$ and $W1 \leq EPS$, in cui W1 denota il valore assunto dalla variabile W alla fine della computazione.
Un dato di test sarà un qualunque assegnamento di valori alle variabili di input del programma che soddisfi il predicato di test considerato (3).

Una rappresentazione schematica dei passi della metodologia di selezione dei dati di test delineata è presentata nella fig. 2.

In questo lavoro presentiamo un generatore di predicati sperimentale, PATHPRED (4), che, dato un cammino di controllo di un sorgente FORTRAN, genera il predicato di test corrispondente. La presentazione verrà effettuata mediante un semplice esempio; a conclusione si prenderanno in considerazione alcuni aspetti riguardanti limitazioni e possibili sviluppi della metodologia proposta, nonchè il rapporto con la tecnica dell'esecuzione simbolica.

2. La generazione dei predicati di test

La generazione dei predicati di test consiste in una esecuzione simbolica "all'indietro" del programma, percorrendo il cammino di controllo scelto, a partire dalla istruzione finale. L'esecuzione ha inizio assegnando alle variabili di output un valore simbolico (che può essere il nome stesso delle variabili), ed opera successive sostituzioni alle variabili di output dei valori simbolici assegnati, valutando le espressioni e i test incontrati.

Consideriamo un semplice esempio (fig. 3), costituito da un programma per il calcolo dell'ascissa del massimo di una funzione FUN, definita in un intervallo (A,B), continua, dotata di un solo massimo nell'intervallo. L'algoritmo, iterativo, si basa su una divisione di (A,B) in 3 intervalli uguali (A,P), (P,Q), (Q,B); i valori della funzione negli estremi P e Q vengono confrontati, e si sceglie come estremo di un nuovo intervallo quello in cui la funzione assume il valore minore. Il valore dell'ascissa viene determinato come punto medio dell'intervallo, con una approssimazione minore o uguale a EPS/2.

```
 1        READ (5,100) A,B,EPS
 2   10   W = B - A
 3        IF (W - EPS) 40,40,50
 4   50   P = A + W/3.
 5        U = FUN (P)
 6        Q = B - W/3.
 7        V = FUN (Q)
 8        IF (U - V) 20,20,30
 9   20   A = P
10        GOTO 10
11   30   B = Q
12        GOTO 10
13   40   MAX = (A + B)/2.
14        WRITE 6,200) MAX
15        STOP
```

Fig. 3

Cammini di controllo di questo semplice programma (cfr.anche fig.1) sono:

c1: 1,2,3,13,14,15
c2: 1,2,3,4,5,6,7,8,9,10,2,3,13,14,15
c3: 1,2,3,4,5,6,7,8,11,12,2,3,13,14,15

dove c2 e c3 sono i due cammini più'brevi' che implichino l'esecuzione del ciclo di iterazione.

Nel caso in cui si scelga c1 come cammino di controllo, la generazione del predicato di test procederà per passi, riassunti nel seguente schema:

P1: MAX=MAXϕ (assegnazione del valore simbolico alla variabile di uscita)

P2: MAX=MAXϕ, MAXϕ=(A+B)/2. (esecuzione 'all'indietro' dell'assegnamento 13)

P3: MAX=MAXϕ, MAXϕ=(A+B)/2.,W$\leq$ EPS (valutazione del test 3)

P4: MAX=MAXϕ, MAXϕ=(A+B)/2., B-A$\leq$EPS (esecuzione 'all'indietro' dell'assegnamento 2)

Il test 3 che si incontra percorrendo all'indietro il cammino c1 deve aver avuto esito positivo, pertanto deve valere la relazione W$\leq$EPS, come riportato al passo P3 dello schema; al passo successivo P4 incontriamo l'assegnamento 2 e rimpiazziamo l'occorrenza della variabile W con il valore che le deve venire assegnato. La procedura di retroesecuzione a questo punto termina, e possiamo interpretare la quarta riga dello schema come la famiglia di proprietà delle variabili del programma, che devono essere contemporaneamente verificate da un insieme di dati di test che provochino una esecuzione del programma secondo il cammino di controllo c1. Delle tre relazioni è solo la terza che determina la classe dei dati di input che garantiscono l'esecuzione del cammino di controllo prescelto, ed è quindi il predicato di test desiderato. Affinchè la esecuzione di prova percorra il cammino di controllo c1, occorrerà perciò trovare tre valori da assegnare alle variabili di input A,B,EPS che soddisfino la relazione:

$$B-A \leq EPS.$$

Poniamo ora di aver commesso un errore di codifica dell'istruzione 4:

4' P=A-W/3.

la retro-esecuzione simbolica relativa al cammino di controllo c2 procederà con i seguenti passi:

P1: MAX=MAXϕ
P2: MAXϕ=(A+B)/2
P3: MAXϕ=(A+B)/2, W$\leq$EPS
P4: MAXϕ=(A+B)/2, B-A$\leq$EPS
P5: MAXϕ=(P+B)/2, B-P$\leq$EPS
P6: MAXϕ=(P+B)/2, B-P$\leq$EPS, U$\leq$V
P7: MAXϕ=(P+B)/2, B-P$\leq$EPS, U$\leq$FUN(Q)
P8: MAXϕ=(P+B)/2, B-P$\leq$EPS, U$\leq$FUN(B-W/3)
P9: MAXϕ=(P+B)/2, B-P$\leq$EPS, FUN(P)$\leq$FUN(B-W/3)
P10: MAXϕ=(A-W/3+B)/2, B-(A-W/3)$\leq$EPS, FUN(A-W/3)$\leq$FUN(B-W/3)
P11: MAXϕ=(A-W/3+B)/2, B-(A-W/3)$\leq$EPS, FUN(A-W/3)$\leq$FUN(B-W/3), W>EPS
P12: MAXϕ=(A-(B-A)/3+B)/2, B-(A-(B-A)/3)$\leq$EPS,

FUN(A-(B-A)/3)≤FUN(B-(B-A)/3),
B-A > EPS

Se risolviamo il sistema di disequazioni:

$$\begin{cases} B-(A-(B-A)/3) \leq EPS \\ B-A > EPS \end{cases}$$

troviamo che non possono essere contemporaneamente soddisfatte le due relazioni

$$\begin{cases} B-A > EPS \\ B-A \leq (3/4)\ EPS, \end{cases}$$

pertanto nel caso preso in esame si ottiene un predicato insoddisfacibile. Il significato da attribuire a questo fatto è duplice:

- o il cammino di controllo selezionato non è percorribile da alcun insieme di dati di input (ovvero il cammino è, in un certo senso, ridondante);
- oppure all'interno del cammino di controllo vi sono errori in qualche istruzione o nel modo in cui il test di controllo è stato realizzato.

Per decidere di quale caso si tratti occorrerà, in questi casi, analizzare il predicato di test ottenuto e il cammino di controllo corrispondente, cercando dei dati di test che provochino l'esecuzione almeno di una parte del cammino desiderato. Nell'esempio trattato, è sufficiente imporre ai dati di input la condizione B-A> EPS affinchè venga percorso il cammino c2 o il cammino c3; sarà così possibile ritrovare l'errore effettuando questa volta una esecuzione reale con dati che soddisfino la condizione imposta.

3. PATHPRED

Il generatore di predicati PATHPRED che è stato sviluppato è un programma FORTRAN che accetta in input una sequenza di istruzioni FORTRAN che rappresentano un cammino di controllo precedentemente selezionato. (La selezione del cammino di controllo, a partire dal sorgente del programma, può essere effettuata automaticamente da appositi programmi, di cui SPRAUT è un esempio (9)).

Un cammino di controllo si presenta in input a PATHPRED come nell'esempio di fig.4, basato sul programma analizzato nei paragrafi precedenti (cammino c2).

```
a         READ (5,100)AAA,BBB,EPS
b    10   WWW = BBB - AAA
c         IF (WWW - EPS) 40,40,50
d    50   PPP = AAA + WWW/3.
e         UUU = FUN(PPP)
f         QQQ = BBB - WWW/3.
g         VVV = FUN(QQQ)
h         IF (UUU - VVV) 20,20,30
i    20   AAA = PPP
l         GOTO 10
m    10   WWW = BBB - AAA
n         IF (WWW - EPS) 40,40,50
o    40   MAX = (AAA + BBB)/2.
p         WRITE (6,200) MAX
q         STOP
```

Fig. 4

PATHPRED analizza il cammino selezionato secondo un algoritmo apposito e stampa in uscita il predicato di test corrispondente:
(BBB - (AAA + (BBB-AAA)/3.))≤EPS
and BBB - AAA > EPS.

L'algoritmo di costruzione del predicato di test è descritto dalla seguente procedura, che analizza il cammino istruzione per istruzione partendo dalla fine:

repeat
    interpreta il test di controllo; individua le variabili coinvolte nel test;
    repeat
    retrosostituisci la condizione di controllo valutando simbolicamente gli assegnamenti incontrati;
    until hai raggiunto l'inizio del programma;
until non ci sono più test di controllo nel cammino.

La procedura "interpreta il test di controllo" consiste nel riconoscimento dell'esito che occorre dare alla condizione di test affinchè il cammino desiderato venga percorso. Nell'esempio, il test alla riga h viene interpretato assegnando alla condizione il valore (UUU - VVV≤ 0,) in quanto l'esito del test(come si vede dal

cammino di fig. 4), ha trasferito il controllo all'istruzione 20. Le variabili coinvolte nel test sono VVV e UUU; la condizione UUU - VVV≤ 0 viene "retrosostituita" assegnando alle sue variabili il valore che esse hanno assunto nelle istruzioni precedenti (ad esempio, al primo passo, essa diventa : (UUU - FUN (QQQ))≤ 0 e così via). Ciò comporta il parsing di tutte le espressioni che contengono variabili che compaiono nella condizione in esame, ed una successiva esecuzione simbolica delle espressioni stesse.

La versione attuale di PATHPRED, costituita da circa 2000 statement FORTRAN, ha carattere sperimentale, in quanto orientata a valutarne le difficoltà di realizzazione; pertanto tratta soltanto IF aritmetici, e non elabora espressioni logiche ed array; inoltre accetta, per semplificare la sezione di parsing, soltanto identificatori di tre caratteri alfanumerici.

4. Conclusioni

Da quanto fin qui visto emergono alcune considerazioni generali riguardo al problema della generazione automatica dei dati di test.

Il primo problema concerne il rapporto con l'esecuzione simbolica. Quest'ultimo è un promettente approccio al testing proposto da King (6), che consiste nel provare un programma assegnando alle variabili di input dei valori simbolici, ed eseguendo il programma con essi secondo le regole del calcolo simbolico algebrico e logico. Alcuni sistemi sviluppati su queste linee guida sono DISSECT (8), che compie una vera e propria esecuzione simbolica su dati di test precedentemente scelti; SELECT (5), che è dotato di un sistema per la scelta dei dati di prova e di un risolutore di predicati e che opera su programmi LISP; il sistema proposto da Ramamoorthy et alt. (7), con caratteristiche simili a quelle di SELECT, operante però su programmi FORTRAN. La generazione dei dati di test secondo il metodo da noi proposto corrisponde a una esecuzione simbolica "all'indietro"; da un altro punto di vista può essere considerata come la scelta delle condizioni cui devono soddisfare i dati simbolici per garantire la terminazione del programma secondo il cammino di controllo selezionato. Secondo l'approccio dovuto a Ramamoorthy et alt. (7), i predicati di test vengono generati con una esecuzione simbolica "in avanti"; dal punto di vista dei risultati i due approcci non presentano tuttavia differenza alcuna.

Un secondo problema riguarda il trattamento degli array; il programma finora sviluppato non tratta array di alcun tipo. Un semplice esempio servirà a illustrare qual è la problematica coinvolta (7):

```
      READ (5,1) I,J
      A(J) = 2
      A(I) = 0
      A(J) = A(J)+1
10    IF (A(J).EQ.3) .....
      .....
```

In una esecuzione simbolica "in avanti", è necessario conoscere il valore di I e J per poter valutare il test 10, e il risultato del confronto logico dipende dall'essere o meno I≠J. In una esecuzione simbolica "all'indietro", scegliendo il cammino selezionato dal verificarsi della condizione A(J).EQ.3, il predicato di test che si ricava sarà:

$$A(J)=2 \text{ and } A(I)=0 \text{ and } 2+1=3,$$

predicato soddisfatto dall'insieme di coppie I,J tali che I≠J. Come si vede, nel caso degli array occorre esprimere il predicato di test in funzione degli indici; il cammino di controllo dipenderà cioè in questo caso non soltanto dai valori delle variabili di input, ma anche dai valori assunti dagli indici degli array nel corso della computazione.

Un terzo problema concerne la risoluzione automatica dei predicati di test per la determinazione di valori da assegnare alle variabili di input. In questo caso occorrerà ricorrere o a dimostratori automatici di teoremi (cfr., per gli aspetti teorici connessi, (10)) o a tecniche di programmazione lineare nel caso in cui il predicato di test sia rappresentato da un sistema di disequazioni lineari.

Un ultimo problema concerne il riconoscimento e la eliminazione dei cammini non percorribili. Una soluzione promettente è quella che introduce dei vincoli sui cammini, ricavabili dalla conoscenza del problema; in tal modo si può sottoporre al generatore di predicati soltanto i cammini sicuramente percorribili.

5. Riferimenti

1. E.F. Jr Miller, PROGRAM TESTING: ART MEETS THEORY, Computer, luglio 1977

2. D.Marini, IL TESTING DEI PROGRAMMI: IL CASO DELLA PROGRAMMAZIONE STRUTTURATA, Giornate di studio su "Programmazione strutturata, esperienze ed orientamenti", Milano, Aerhotel, giugno 1976

3. D.Marini, M.Ornaghi, UN APPROCCIO ALLA TEORIA DEL TESTING DEI PROGRAMMI, Congresso AICA, Genova, 1975

4. G.Ferraro, ASPETTI TEORICI E PRATICI DEL TESTING: LA GENERAZIONE AUTOMATICA DEI DATI DI TEST, Tesi di laurea in Fisica, Università di Milano, 1978

5. R.S.Boyer et al., SELECT - A FORMAL SYSTEM FOR TESTING AND DEBUGGING PROGRAMS BY SYMBOLIC EXECUTION, Proceedings of the International Conference on Reliable Software, Los Angeles, 1975

6. J.C.King, A NEW APPROACH TO PROGRAM TESTING, Proceedings of the International Conference on Reliable Software, Los Angeles, 1975

7. C.V.Ramamoorthy et al., ON THE AUTOMATED GENERATION OF PROGRAM TEST DATA, IEEE Transactions on Software Engineering, Vol.SE-2, n.4 (aprile 1976)

8. W.E.Howden, DISSECT - A SYMBOLIC EVALUATOR AND PROGRAM TESTING SYSTEM, IEEE Transactions on Software Engineering, Vol. SE-4, n.1 (gennaio 1978)

9. D.Pastori, SPRAUT: UNO STRUMENTO PER LA GUIDA ALLA STESURA DI DATI DI TEST, Tesi di laurea in Fisica, Università di Milano, 1974

10. C.L.Chang, R.C.T.Lee, SYMBOLIC LOGIC AND MECHANICAL THEOREM PROVING, Academic Press, 1973

Il presente lavoro è inquadrato nel CP Project di collaborazione tra l'Università di Milano - Istituto di Cibernetica e la Honeywell Information Systems Italia.

UNO STRUMENTO PER LA GESTIONE DI ARCHIVI DI TEST

A. Jemoli

H.I.S.I., Pregnana Milanese

1. INTRODUZIONE

In questa nota viene descritto un semplice sistema per la gestione di archivi di test di prodotti software, realizzato sul Livello 62. Esso si integra con una serie di prodotti già descritti altrove [1], [2], [3], il cui scopo complessivo è quello di contribuire a rendere automatica e centralizzata l'attività di qualificazione del software.

Qualificazione automatica, perchè queste procedure consentono di scegliere tra tutti i test realizzati quelli che si desiderano lanciare, metterli in forma lanciabile, lanciarli, registrarne i risultati ed infine stampare gli opportuni rapporti di certificazione.
Tutto ciò con un intervento minimo da parte dell'operatore.

Qualificazione centralizzata, in quanto queste procedure consentono di archiviare su dischi tutti i test di qualificazione, e di gestirli in modo standard.

Dopo alcune indicazioni su che cosa si intende, in questa sede, per 'test' (§ 2) e sulla struttura generale di un 'archivio di test' (§3), si descrivono le principali caratteristiche del prodotto (§4 e § 5).

2. CONCETTI GENERALI SUI TEST

Con il termine test, nella presente nota, intendiamo:

- una sequenza di frasi JCL (o di dati) compresa fra una $JOB e una $ENDJOB, che costituisca, dal punto di vista logico,
- una unità minima di lancio, e che possa considerarsi
- indipendente da tutti gli altri test (nel senso che il suo esito non è condizionato dall'esito di altri test), e sia, inoltre,
- autosufficiente (in grado cioè di preallocarsi i file necessari e deallocarli alla fine, di prepararsi i dischi di lavoro richiesti, ecc.).
- Lo scopo del test è di qualificare una 'sola' funzionalità ben definita, per localizzare meglio un eventuale malfunzionamento.
- Il report di uscita (ci si riferisce qui al report del TRP, vedi oltre) deve essere uno solo e relativo alla funzionalità esercitata.
- Ogni test è individuato nell'archivio da una chiave, che identifica:
  a) il nome del 'complex' di sistema che il test ha lo scopo di qualificare (5 crt.);
  b) il nome della catena di appartenenza (7 crt., vedi oltre);
  c) il nome del test (8 crt.).

Il nome del complex deve essere un nome standard nell'ambito del sistema e il nome della catena può essere ϕ; in questo caso, il test non è collegato ad altri e costituisce una catena a sè stante.

Per catena si intende, qui, un insieme di test, destinati a qualificare funzionalità 'omogenee'. Il concetto di catena viene introdotto al solo scopo di semplificare la gestione e il lancio dei test: il risultato di un test di una catena non deve influenzare il risultato di test successivi appartenenti alla stessa o ad altre catene.

3. FILOSOFIA E CONTENUTO DELL'ARCHIVIO

Per archivio intendiamo:

- Archivio generale: un luogo in cui sono depositate le norme operative dei vari test, i dischi in cui essi sono archiviati, i dischi di lavoro necessari alla qualificazione ed eventuali schede, nastri, cassette, ecc.

- Archivio delle norme operative: uno o più manuali in cui sono descritte le funzionalità di ciascun test, le aree coperte e il suo funzionamento, oltre alle norme per l'operatore che ne seguirà il lancio. Questo o questi volumi devono avere un indice organizzato per nome-di-complex/nome-di-catena/nome-di-test.

- Archivio dei test (fig. 1): uno o più dischi, in ciascuno dei quali vi sono:
  a) un file direttrice (FTOC: File Table of Content) del 'file di JCL', contenente, a fronte di ogni chiave, l'indirizzo in cui si trova la sequenza di JCL relativa a quel test;
  b) un file di JCL (ARCHIVIO), contenente le frasi di JCL e i dati per il lancio di tutti i test registrati nella direttrice;
  c) un file anagrafico (gestito da TRP), relativo a tutti i test registrati nella direttrice;
  d) un file di report (gestito da TRP), relativo anch'esso a tutti i test registrati nella direttrice

e, opzionalmente (in quanto alcuni test comprendono solo load module già contenuti nel system disc da qualificare):

  e) una libreria di load module, contenente tutti i LM relativi ai test registrati nella direttrice;
  f) una libreria di compilation unit, contenente tutte le CU richiamate dai test registrati nella direttrice (saranno presenti ove nel JCL vi siano delle $LINK);
  g) una libreria di source unit, contenente i sorgenti richiamati dai test registrati nella direttrice (saranno presenti ove nel JCL vi siano richiami a compilatori).

Il 'file anagrafico' e il 'file di report' sono gli archivi gestiti da TRP ('Test Reporting Package', vedi [1] e [2]), un sistema che permette di stampare vari tipi di rapporti sullo stato di certificazione di un prodotto software (quali test di specificate caratteristiche sono stati eseguiti e con quale esito, quali test non sono stati eseguiti, ecc.). Per questo, TRP dispone di un 'file anagrafico', contenente opportune informazioni anagrafiche sui test (nome, funzionalità coperte, release, ecc.) e un 'file di report', in cui i test stessi registrano, al momento della loro esecuzione, varie informazioni sulla configurazione di sistema (H/W e S/W) in cui sono stati lanciati, e il loro esito ('successo', 'fallimento', 'esito indefinito'). Queste informazioni vengono registrate da apposite macro ($JSUCCESS, $JFAIL, ecc.), fornite da TRP, che il progettista dei test provvede a inserire in posizione opportuna nel test stesso (in una catena di frasi JCL, nel codice di un programma assembler, ecc.).

4. OPERAZIONI POSSIBILI SULL'ARCHIVIO

Le librerie di LM, CU, SU sono standard: su di esse sono possibili tutte le operazioni consentite dalla Library Maintenance di sistema.

Per il file anagrafico e di report, sono possibili tutte le operazioni consentite dal TRP.

Per i file FTOC e ARCHIVIO sono possibili:

a) il caricamento, che avviene tramite ordine da consolle e successiva lettura del pacco di schede da caricare. Ogni pacco di schede inizia con la scheda:

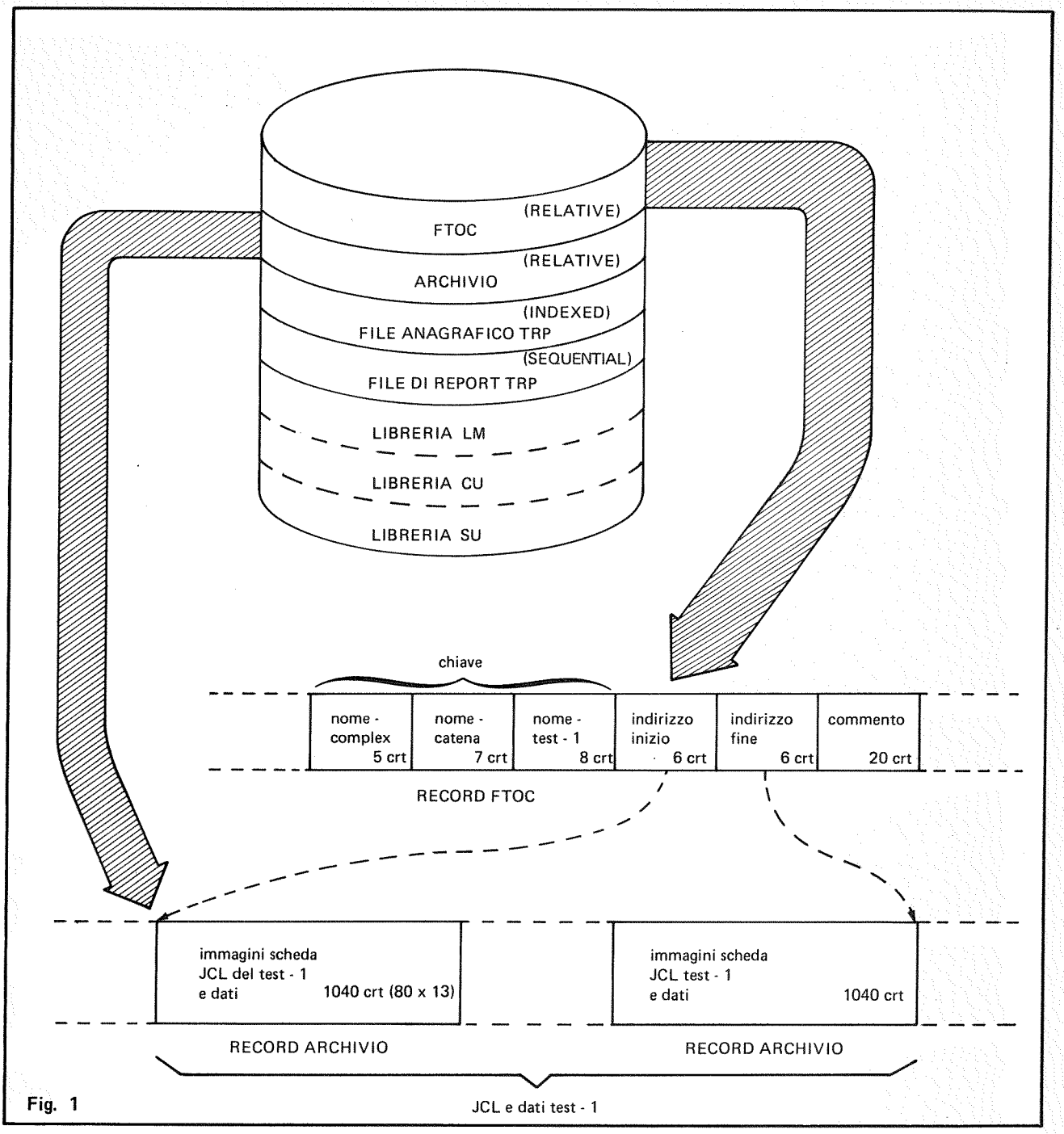


Fig. 1

#nome-complex, nome-catena, nome-test

a cui seguono schede contenenti le frasi di JCL e gli eventuali dati del test in oggetto;

b) la lista del contenuto della direttrice o del file di JCL;

c) la cancellazione di un test, ottenuta ponendo a ½ la relativa entry della direttrice;

d) l'aggiornamento di un test, che avviene accodando il nuovo JCL al file, e inserendo i nuovi indirizzi nella entry relativa della direttrice;

e) il compattamento della direttrice e del file di JCL, che permette l'eliminazione delle entry non più utilizzate, e il JCL non più puntato da entry valide.

Le operazioni b), c), d), e) sono effettuabili anche mediante richiesta da consolle.

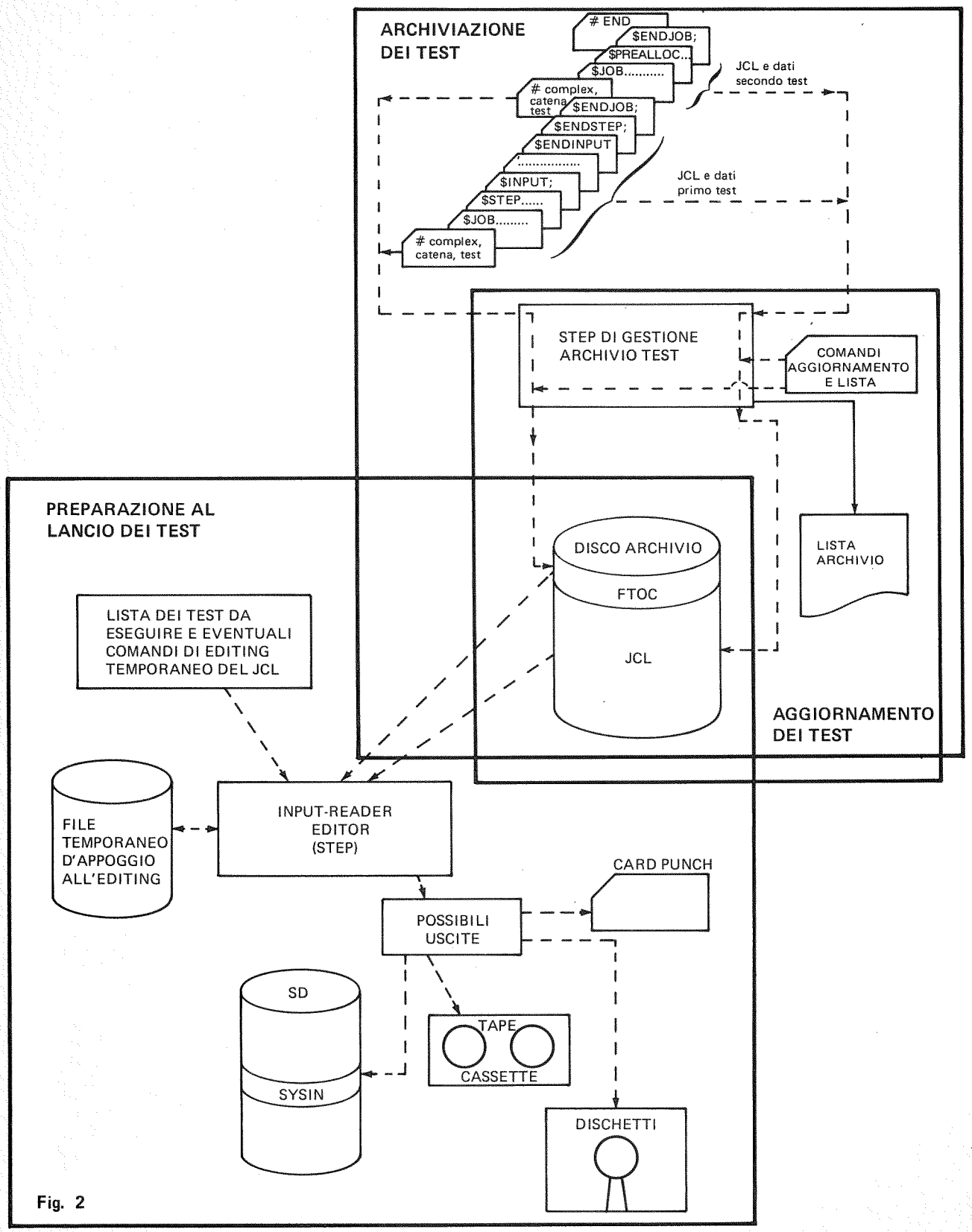

ARCHIVIAZIONE
DEI TEST
END
$ENDJOB;
$PREALLOC...
$JOB.....
complex,
catena
test
$ENDJOB;
$ENDSTEP;
$ENDINPUT
.....
$INPUT;
$STEP.....
$JOB.....
complex,
catena, test
JCL e dati
secondo test
JCL e dati
primo test
STEP DI GESTIONE
ARCHIVIO TEST
COMANDI
AGGIORNAMENTO
E LISTA
PREPARAZIONE AL
LANCIO DEI TEST
DISCO ARCHIVIO
FTOC
JCL
LISTA
ARCHIVIO
LISTA DEI TEST DA
ESEGUIRE E EVENTUALI
COMANDI DI EDITING
TEMPORANEO DEL JCL
AGGIORNAMENTO
DEI TEST
FILE
TEMPORANEO
D'APPOGGIO
ALL'EDITING
INPUT-READER
EDITOR
(STEP)
CARD PUNCH
POSSIBILI
USCITE
SD
SYSIN
TAPE
CASSETTE
DISCHETTI

Fig. 2

INDIRIZZO DEL DESTINATARIO:

Ragione Sociale

* Indirizzo Spedizione

C.A.P. Città

Nominativo (Cognome - Titolo - Nome)

SETTORE D'ATTIVITA' DELL'ENTE: .....

SETTORI DI INTERESSE (Specificare il tipo):

- ☐ software di base .....
- ☐ applicazioni edp .....
- ☐ applicazioni numeriche .....
- ☐ controllo di processo .....
- ☐ altro .....

- ☐ grossi sistemi .....
- ☐ medi sistemi .....
- ☐ minicomputer .....
- ☐ microprocessor .....
- ☐ altro .....

- ☐ project management .....
- ☐ analisi di sistemi .....
- ☐ progettazione di sistemi .....
- ☐ programmazione .....
- ☐ testing & quality assurance .....
- ☐ manutenzione .....
- ☐ valutazione delle prestazioni .....
- ☐ formazione .....
- ☐ altro .....

EVENTUALE NOMINATIVO DI PERSONA INTERESSATA A RICEVERE **NOTE DI SOFTWARE**:

Ragione Sociale

* Indirizzo Spedizione

C.A.P. Città

Nominativo (Cognome - Titolo - Nome)

SETTORE DI ATTIVITA' DELL'ENTE: .....

* Se si tratta di indirizzo privato, farne esplicita menzione.

SU QUALI ARGOMENTI, NELL'AMBITO DELL'INGEGNERIA DEL SOFTWARE, RITERREBBE PIU' UTILE CHE LA RIVISTA PUBBLICASSE CONTRIBUTI TECNICI O NUMERI SPECIALI MONOGRAFICI?

.....

.....

.....

ALTRI COMMENTI, SUGGERIMENTI, PROPOSTE:

.....

.....

.....

.....

.....

E' DISPONIBILE AD INVIARE LAVORI PER LA PUBBLICAZIONE SU **NOTE DI SOFTWARE** (contributi tecnici, tutorial, rassegne, ecc.)?

☐ SI, sui seguenti argomenti .....

.....

☐ NO

DESIDERA PROPRORRE, FIN DA ORA, UN LAVORO SPECIFICO?

☐ SI: titolo ed autori (indicativi) .....

.....

.....

☐ NO

INTENDIAMO PUBBLICARE, SU UNO DEI PROSSIMI NUMERI DI **NOTE DI SOFTWARE**, IL CATALOGO DI UNA IMMAGINARIA "BIBLIOTECA ESSENZIALE" SULL'INGEGNERIA DEL SOFTWARE NEI SUOI VARI ASPETTI, COSTRUITA SULLA BASE DELLE INDICAZIONI DEI LETTORI DELLA RIVISTA. LE CHIEDIAMO PERTANTO LA SUA COLLABORAZIONE NEL RISPONDERE ALLE SEGUENTI TRE DOMANDE.

A – PUO' ELENCARE I TITOLI DEI LIBRI (italiani o stranieri) CHE TALE BIBLIOTECA ESSENZIALE A SUO PARERE DOVREBBE POSSEDERE? (massimo 15 titoli)

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

B – PUO' ELENCARE I TITOLI DI LAVORI SCIENTIFICI (articoli, rapporti tecnici, ecc.) CHE TALE BIBLIOTECA ESSENZIALE A SUO PARERE DOVREBBE POSSEDERE? (massimo 15 titoli)

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

C – PUO' ELENCARE I TITOLI DELLE RIVISTE PERIODICHE CHE A SUO PARERE TALE BIBLIOTECA ESSENZIALE DOVREBBE POSSEDERE? (massimo 5 testate)

.....

.....

.....

.....

.....

.....

.....

.....

PIEGARE

Affrancatura
L. 170

NOTE DI SOFTWARE

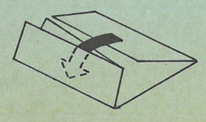
HONEYWELL
INFORMATION SYSTEMS ITALIA

via ai laboratori Olivetti

20010 PREGNANA MILANESE
(MILANO)

PIEGARE

CHIUDERE COME IN FIGURA



5. PREPARAZIONE E LANCIO DEI TEST

La procedura di preparazione consente di scegliere fra tutti i test presenti in archivio quelli elencati in una lista fornita da schede o da consolle, e di creare un input-stream contenente tutte le frasi JCL (e i dati) dei test richiesti, cui saranno accodate opportune frasi JCL che richiamino il TRP per la stampa dei report.

I file di input-stream possono essere generati su cassette, dischetti o schede per input-reader.

La procedura di lancio dei test consiste semplicemente nell'eseguire l'input-stream generato. Poichè gli ultimi due statement di tale input-stream sono la macro di richiamo del TRP e la $ENDJOB, si avrà, prima del termine dell'attività, la stampa dei quality report dei soli test lanciati.

6. RIFERIMENTI

[1] 'Test Reporting Package User Guide', documento interno H.I.S.I., SE 0T20, novembre 1976

[2] S. Farnedi, C. Poggiali, R. Polillo, A. Vignoni, 'TRP: un sistema per la gestione dei risultati delle prove di un sistema operativo', Atti del Convegno AICA, Milano, 1976

[3] JMLOAD, Functional Specs., doc. interno HISI.

SPECIALIZZAZIONE DI UN COMMAND LANGUAGE IN AMBIENTE BATCH REMOTO

G. Galdi, G. Petraglia, G. Vildacci (+)

1. INTRODUZIONE.

Lo sviluppo di una nuova architettura dell'elaboratore non legata ad una struttura gerarchica monoblocco, ma con disegno modulare, anche di tipo multiprocessor, e lo sviluppo di periferiche di tipo intelligente, ha portato ad una profonda revisione del tipo di software da installare sulle macchine e ad una evoluzione delle modalità operativo-gestionali. Questa distribuzione di risorse h/w, se da una parte permette di creare un sistema informativo di tipo distribuito, dall'altra necessita di una logica s/w che coordini e controlli il funzionamento delle varie unità.

Poichè per macchine di medie dimensioni le funzioni di I/O rappresentano pur sempre un limite (a meno di non aumentare di molto i costi), è opportuno sfruttare la risorsa linee-di-comunicazione non solo per la gestione, ad es., di dati in tempo reale, ma anche per tutta una serie di funzioni che potremmo raggruppare sotto il nome di gestione di procedure da remoto.

Il command language qui esaminato, JCL del Livello 62 HONEYWELL, possiede caratteristiche di 'programmabilità' tali che, opportunamente sviluppate in termini di strutture di controllo, tipiche della programmazione strutturata, permettono di realizzare strumenti di gestioni di procedure da remoto.

Col presente articolo abbiamo voluto verificare questa possibilità su una tipica procedura (gestione dei programmi), che non è l'unica prevedibile, ma è comunque una delle più onerose per i centri di elaborazione dati.

(+) Ist. Scienze dell'Informazione,
Università di Salerno.

La possibilità di gestire da remoto questa procedura dimostra che un command language sufficientemente implementato rende possibile la realizzazione di una più organica gestione delle procedure e delle fasi di lavoro.

2. GESTIONE PROGRAMMI DA REMOTO

Le funzioni di gestione programmi sono compilazione, catalogazione sorgenti, correzione sorgenti, allocazione e generazione di flussi di dati-prova, debugging e testing dei programmi.
Queste sono le funzioni base svolte da un reparto di programmazione.

La possibilità di eseguire il lavoro di progettazione in remoto, permette di svincolare le attività del reparto di progettazione da quelle del reparto operativo.

I vantaggi di questa soluzione sono due. Il primo è la possibilità di liberare risorse, quali stampante e unità di system input (lettore di schede, diskette, tape-cassette), lasciandone il pieno sfruttamento alla funzione operativa centralizzata; il secondo è la diretta interazione programmatore-computer per tutte quelle funzioni di manutenzione dei programmi che altrimenti richiederebbero il passaggio attraverso il reparto operativo, con possibilità di fenomeni di ingorgo di quest'ultimo e comunque con un ritardo nei tempi di risposta.

Per rendere possibile la gestione dei programmi da remoto su elaboratori che non dispongono della possibilità di governare il sistema da remoto (Remote Job Entry), è necessario costruire un'interfaccia tra il programmatore ed il JCL della macchina, che permette di lanciare compilatori, utilities, ..., da terminale.

Per garantire la massima semplicità e rapidità di uso e non appesantire il lavoro del programmatore, é opportuno che l'interfaccia sia specializzata, che consenta cioé di richiedere le prestazioni del sistema con comandi semplici ed immediati.

Un esempio di linguaggio di comando di una tale interfaccia é l'RBL, descritto nel prossimo paragrafo.

3. MINILINGUAGGIO RBL

Il minilinguaggio RBL é formato da una serie di comandi operativi ognuno dei quali richiama un insieme di prestazioni del s/w standard della macchina.

La sintassi di RBL in BNF é la seguente:

```
<prog. RBL> ::=<corpo><new line> @END
<corpo> ::=<direttiva><corpo>|<direttiva>
<direttiva> ::=<comando><nl><linee dati>
                <nl>|<comando><nl>
<comando> ::= @COB[, nome-prog[,
              nome-lib]]|@DATI|@JCL|
              @OUT|@NRP.
<linee-dati> ::=<linea-dati><nl><linee dati>
              |<linea dati>
<nome-prog> ::= stringa-8
<nome-lib>  ::= stringa-8
<linea-dati> ::= stringa-80
<nl>::= salto a nuova riga
```

- stringa-80 é una stringa di al massimo 80 caratteri che non contiene il carattere @ ;
- stringa-8 é una stringa di 8 caratteri alfanumerici di cui il primo dev'essere alfabetico.

Il significato dei comandi é il seguente:

@OUT
permette di ottenere una riedizione dello ultimo report ricevuto;

@NRP
specifica che l'utente non desidera ricevere il report;

@DATI
tutte le linee comprese tra questa direttiva e la successiva vengono memorizzate su file sequenziale organizzato ad immagine scheda che può essere usato come input per eventuali elaborazioni richieste successivamente;

@COB[nome-prog [LIB=(descr.lib)]]
questa direttiva consente la compilazione di un programma sorgente COBOL.
Le linee di codice devono seguire immediatamente la direttiva oppure il testo deve risiedere nella libreria specificata.
Il formato del codice deve essere quello standard dei programmi COBOL e nome-prog deve essere lo stesso del paragrafo PROGRAM-ID;

@FOR[nome-prog [LIB=(descr. lib)]]
consente la compilazione di un programma sorgente FORTRAN;

@JCL
consente di introdurre come statements successivi una sequenza di ordini espressi direttamente nel linguaggio JCL della macchina, limitatamente alle operazioni di lancio programmi, allocazione di files, generazione di dati-prova;

@END
chiude obbligatoriamente una sequenza di direttive RBL.

4. APPROCCIO RBE.

Un prototipo di interfaccia per la gestione di programmi da remoto é RBE (Remote Batch Entry) che é stato realizzato su elaboratore L62/40 con 144 Kbytes.

RBE utilizza stazioni remote costituite da un video terminale, due unità a cassette e stampante seriale. Inoltre richiede le seguenti librerie su disco:

- una libreria di load modules che contiene il s/w RBE;
- tre work files sequenziali chiamati RBE@DATI, RBE@COMP, RBE@JCL, di cui solo il primo é visibile all'utente;
- una libreria di procedure (non visibile all'utente);
- un file di report;
- un file che contiene gli statements JCL per il lancio di RBE (input stream).

RBE é costituito da tre blocchi logici (fig. 1). Il primo modulo (COLLEZIONE) interagisce con la stazione remota, avviando il

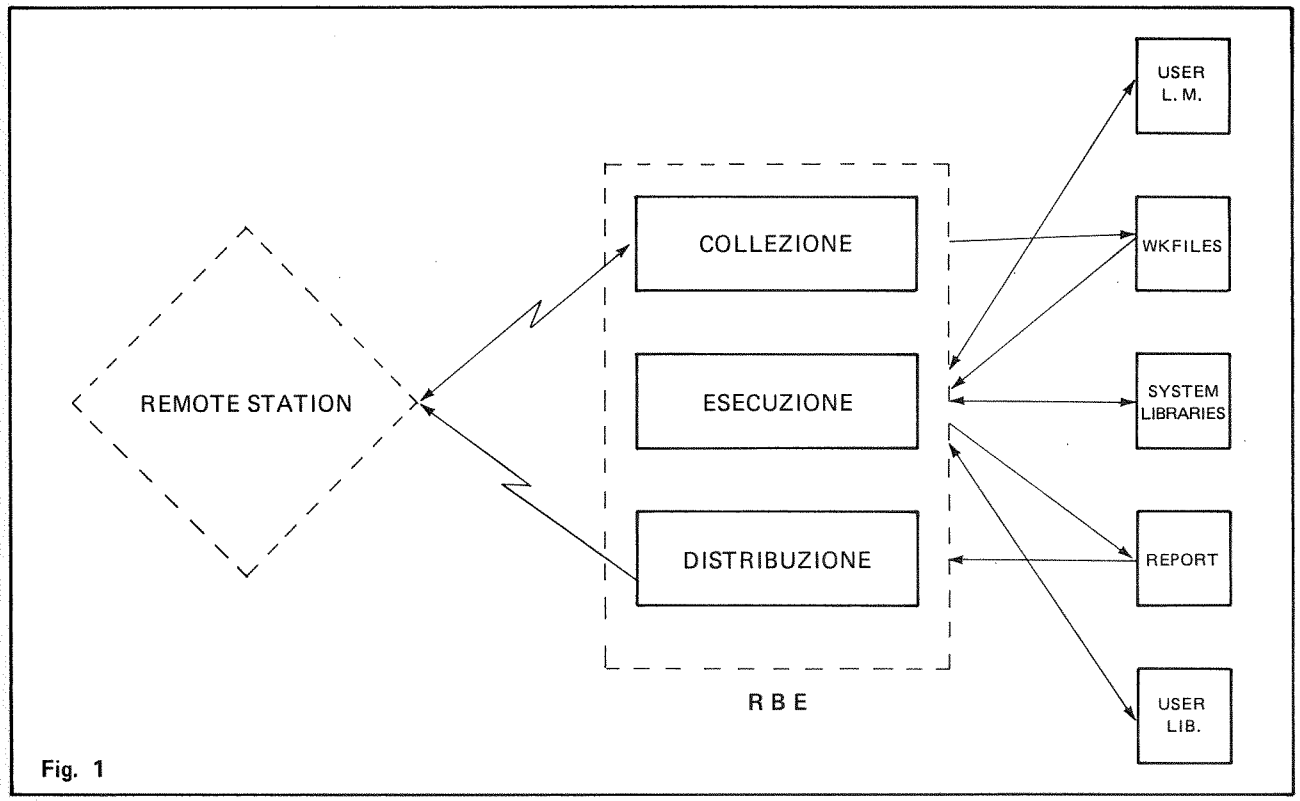


Fig. 1

colloquio e accettando ordini redatti nel mini linguaggio specifico (RBL) e memorizzando temporaneamente i dati destinati ad essere utilizzati dal blocco successivo.
L'unità di input può essere selezionata durante la fase iniziale del collegamento. L'utente può scegliere la tastiera oppure una delle due unità a tape cassette.

Il blocco ESECUZIONE può eseguire un certo insieme di funzioni sui dati o sui programmi appena introdotti o su quelli già residenti in libreria.
Questo blocco é controllato dal precedente attraverso il passaggio di parametri che contengono la traduzione dei comandi RBL.
E' possibile compilare e produrre i load modules di programmi COBOL o FORTRAN, eseguire statements JCL, lanciare programmi, operare sulle librerie utente.

I reports prodotti in questa fase vengono memorizzati sul report file e passati al blocco DISTRIBUZIONE.
Quest'ultimo blocco provvede alla distribuzione dei reports generati durante la fase di esecuzione.

L'intera procedura é scritta nel linguaggio JCL della macchina; utilizza gli statements e i macro-statements standard più due programmi (STEP) scritti appositamente:
lo STEP COLLECTION e lo STEP DISTRIBUTION.

Diamo ora una breve descrizione dei blocchi logici.

Blocco COLLEZIONE. E' un programma JCL (fig. 2), formato da dichiarative di variabili di controllo (K1, K2,...), da uno step (COLLECTION) e da una serie di test ($WHEN) che controllano la terminazione anormale dello step e la richiesta di terminazione di RBE. Le informazioni su questi eventi vengono passate agli altri blocchi attraverso il meccanismo delle macro $SAVECV e $RESTCV che simulano il passaggio di parametri tra blocchi JCL.
Anche lo step COLLECTION attribuisce valori ad alcune variabili di controllo attraverso il meccanismo chiamato JOB LINKAGE caratteristico del s/w dell'L62, che permette di modificare all'interno di uno step utente le variabili definite nel job.

Lo step COLLECTION contiene una sezione di controllo, una di comunicazione ed un interpretatore del linguaggio RBL che fornisce anche segnalazioni di errore nel caso che le frasi RBL non siano state scritte correttamente.

Blocco ESECUZIONE. E' costituito da diversi segmenti ognuno dei quali comanda la esecuzione di una delle funzioni descritte da un ordine RBE. In fig. 3 é illustrato il meccanismo di selezione dei comandi. Nel blocco esistono procedure JCL (FASTCOB, ecc., ...) predefinite, costituite da una chiamata del rispettivo compilatore seguita dalla chiamata del programma linker che genera i load modules eseguibili.

Blocco DISTRIBUZIONE. Si tratta ancora di un programma JCL che 'importa' il valore di una variabile controllata dallo step COLLECTION attraverso l'uso di $RESTCV. Questo valore viene controllato per decidere se deve essere trasmesso o meno il report prodotto durante la fase di esecuzione del blocco ESECUZIONE. In caso affermativo, viene eseguito lo step DISTRIBUTION, in caso contrario si salta direttamente allo step RESET che azzera tutte le variabili temporanee.

5. ESEMPI DI APPLICAZIONI DI RBE.

Nelle figg. 4, 5, 6 riportiamo tre tipici esempi di applicazione di RBE:

- una compilazione di un programma COBOL;
- la correzione di un programma sorgente;
- una fase di testing di un programma in cui i dati-prova sono generati automaticamente dalla macro $TDG.

Nelle tre figure distinguiamo tre tipi di messaggio, e precisamente:

- messaggio proveniente dal computer e che non richiede risposta:
  prefisso ♭****;
- messaggio proveniente da computer e che richiede risposta:
  prefisso ?****;
- messaggi digitati dall'operatore:
  senza prefisso.

Negli esempi é riportata la sequenza dei messaggi come apparirebbe al terminale. Per il caricamento dei dati si fa uso del file RBE@DATI, accessibile attraverso il comando @DATI.

6. RIFERIMENTI.

[1] G. GALDI-G. PETRAGLIA: Progettazione modulare nella gestione di un sistema di archivi. Aica 77.

[2] M. DI LORENZO: Remote Batch Entry. Tesi di laurea I.S.I. Salerno.

[3] F. C. HALL: RTE (Remote Text Editor).

[4] H. J. WEEGENAAR-D.K. WIELENGA: Toward a generalized command language for job control. Proc. of the IFIP Working Conference on Command Languages. Sweden 1974.

[5] HONEYWELL ISI: GCOS-62: System Control Manual.

[6] K. BANDAT: Concepts for a generalized command language. Proc. of the IFIP Working Conference on Command Languages. Sweden 74.

```
$JOB COLLECT;
        $DCV K1,...                     definizioni di variabili
        $DCV K2,...                     di controllo
        ...
        $STEP COLLECT;
        ...
        $ENDSTEP ;
        $SAVECV K1,TO=TEMP01;           salva i valori di K1 e K2
        $SAVECV K2,TO=TEMP02;
        ...
$ENDJOB ;
        $REPORT RB1,...                 viene selezionato il file di report
```

fig. 2

```
$JOB EXEC;
        $DCV JCOMP,...
        $DCV JNOME,...
        $RESTCV JCOMP,FROM=TEMP02;      -assegna alla variabile JCOMP
                                         il valore contenuto in TEMP02
        $JUMP %JCOMP;                   -salta al valore di JCOMP('COB','FOR',..)
COB     $RESTCV JNOME,FROM=TEMP01;
        $RESTCV JLIB,FROM=TEMP09;
        $EXECUTE FASTCOB,...
        $JUMP NOT;
FOR     ...
NOT     $CONTINUE ;
        $RESTCV JSCL,FROM=TEMP07;
        $WHEN JSCL,=,' ',JUMP=GOON;     -salta se la condizione è vera
        $STORE NAME=RBEPROC,...         -JSCL é diversa da b/ se é richiesta
                                         l'esecuzione di statements JCL
        $EXECUTE NAME=RBEPROC,...       gli statements introdotti da terminale
                                         vengono caricati in libreria (STORE) e
GOON    $CONTINUE ;                      quindi eseguiti (EXECUTE)
$ENDJOB ;
```

fig. 3

```
 **** START OF RBE:COLLECTION
?**** INPUT FROM? REPLY (TCU00N,KEYBOARD,TERMINATE)
KEYBOARD
 **** READY
@DATI
 S00230
...                                     digitazione statements di edit
//
@JCL
$EDIT NAME=PROG1,SOURCE=(FILE=RBE@DATI,MEDIA=RBE),...
@END
 **** END OF COLLECTION
```

fig. 4

```
 **** START OF RBE : COLLECTION
?**** INPUT FROM? REPLY (TCU00N,KEYBOARD, TERMINATE)
KEYBOARD
 **** READY
@COB, PROG
000010 IDENTIFICATION DIVISION.
000020 PROGRAM-ID. PROG.
...
001120      STOP RUN.
@END
 **** END OF COLLECTION
 **** START OF DISTRIBUTION
?**** OUTPUT TO? REPLY (TCU00N, PRINTER)
PRINTER
        ii viene eseguita la stampa del report
```

fig. 5

```
 **** START OF RBE: COLLECTION
?**** INPUT FROM? REPLY (TCU00N,KEYBOARD, TERMINATE)
KEYBOARD
 **** READY
@DATI                                       digitazione degli statements  in input
...                                         al test data generator
@JCL
$PREALLOC FILE=PROVA1,...
$TDG SOURCE=( FILE=RBE DATI,...             generazione di un file di dati-prova
     OUTFILE=(FILE=PROVA1,...
$PREALLOC FILE=PROVA2,...
$STEP PROG2;                                lancio del programma da collaudare
        $ASSIGN F1,PROVA1,...
        $ASSIGN F2,PROVA2,...
$ENDSTEP ;
$PRFILE FILE=PROVA2,...                     stampa del file di output
@END
 **** END OF COLLECTION
 **** START OF DISTRIBUTION
?**** OUTPUT TO? REPLY (TCU00N, PRINTER)
PRINTER
...
 **** END OF DISTRIBUTION
 **** START OF RBE: COLLECTION              nuova fase di collezione
?**** INPUT FROM? REPLY (TCU00N,KEYBOARD, TERMINATE)
TERMINATE                                   richiesta di terminazione
 **** END OF RBE
```

fig. 6

BIBLIOGRAFIA

IL LINGUAGGIO PASCAL NELLA LETTERATURA

G. Castelli
Ist. di Cibernetica - Univer. di Milano

Il presente contributo è nato dall'esigenza di poter disporre di un elenco il più possibile esaustivo ed eventualmente corredato da brevi note, di quanto è stato scritto finora sul linguaggio di programmazione Pascal. La diffusione, se non commerciale almeno sperimentale, di Pascal ha infatti raggiunto livelli ragguardevoli; e l'interesse continua ad essere vivo, a giudicare anche dal numero di linguaggi da esso derivati.

I lavori sono stati raggruppati per gran di classi di argomenti:

a) Descrizioni generali del linguaggio
b) Commenti e proposte di miglioramento
c) Definizioni formali
d) Compilatori ed implementazioni del linguaggio
e) Confronti con altri linguaggi
f) Introduzioni alla programmazione in Pascal
g) Standard di programmazione in Pascal
h) Pascal e la didattica
i) Applicazioni e strumenti.

All'interno di ciascun gruppo, i lavori sono ordinati cronologicamente. Le note tra virgolette sono tratte dai lavori stessi. Alcuni titoli non sono commentati, poichè i lavori relativi non sono risultati disponibili, e sono stati inclusi per completezza.
Nella compilazione, sono state esaminate le seguenti riviste:

Communications of the ACM
SIGPLAN Notices (ACM)
Transactions of the IEEE on Software Engineering
Software-Practice and Experience
Acta Informatica

e vari atti di congressi e seminari.
Alcuni articoli sono stati pubblicati sotto forme diverse (rapporti interni o pubblicazione su riviste): i riferimenti sono di solito fatti a quest'ultima forma.

a) Descrizioni generali del linguaggio

● Wirth N., THE PROGRAMMING LANGUAGE PASCAL AND ITS DESIGN CRITERIA, NATO Conference on Software Engineering, Roma, ottobre 1969.

● Wirth N., THE PROGRAMMING LANGUAGE PASCAL, Acta Informatica, vol.1, n.1, 1971, 35-63.

"A programming language called Pascal is described which was developed on the basis of Algol 60. Compared to Algol 60 its range of applicability is considerably increased due to a variety of data structuring facilities. [...] emphasis was placed on keeping the number of fundamental concepts reasonably small, on a simple and systematic language structure, and on an efficient implementability. A one-pass compiler has been constructed for the CDC-6000 computer family; it is expressed entirely in terms of Pascal itself."

● Jensen K., Wirth N., PASCAL - USER MANUAL AND REPORT, Lecture Notes in Computer Science n.18, Springer-Verlag, 1974, pp.VII-170.

"This manual is directed to those who have previously acquired some programming skill. The intention is to provide a means of learning Pascal without outside guidance. It is based on The programming language Pascal (Revised report) - the basic definition of Pascal and concise reference manual for the experienced Pascal programmer.[...] " Contiene anche la documentazione dell'implementazione di Pascal su un CDC 6000, con le aggiunte apportate allo standard e l'uso del compilatore sotto il sistema operativo SCOPE.

● Wirth N., AN ASSESSMENT OF THE PROGRAMMING LANGUAGE PASCAL, Proceedings of the International Conference on Reliable Software 1975, Sigplan Notices (ACM), vol.10, n.6 (giugno 1975), 23-30.

"The programming language Pascal is assessed in the light of reliable programming and with the background of five years of experience with the language. Some features are selected to point out remaining problems, either inherent or specific, from which some guidelines for the design or choice of languages for reliable programming are derived. Among the discussed features are the concept of data type, the sequential file structure, and the type union."

● Hansen, P.B., Hartmann A., SEQUENTIAL PASCAL REPORT, Information Science, California Institute of Technology.

b) Commenti e proposte di miglioramento

● Habermann A.N., CRITICAL COMMENTS ON THE PROGRAMMING LANGUAGE PASCAL, Acta Informatica, vol.3, n.1, 1973, 47-58.

E' l'articolo più critico nei confronti di Pascal e che ha suscitato vivaci reazioni da parte di numerosi altri autori.

"The programming language Pascal is claimed to be more suitable than other languages for teaching programming as a systematic discipline. However, an investigation of the reports on the Pascal language reveals that it suffers as much from ill-defined constructs [...] Problems are caused by the confusion of ranges, types and the structures and by the phenomena associated with GO TO statements."

● Lecarme O., Desjardins P., REPLY TO A PAPER BY A.N. HABERMANN ON THE PROGRAMMING LANGUAGE PASCAL, Sigplan Notices (ACM), vol.9, n.10 (ottobre 1974), 21-27.

Si tratta di una risposta all'articolo di Habermann 'Critical comments on the programming language Pascal'. Sono discusse nei loro vantaggi e svantaggi la presenza di certe frasi (es., GOTO), l'assenza di alcune caratteristiche (es., array variabili) e alcune scelte di base di Pascal (es., struttura a blocchi).

● Lecarme O., Desjardins P., MORE COMMENTS ON THE PROGRAMMING LANGUAGE PASCAL, Acta Informatica, vol.4, n.3, 1975, 231-244.

Si tratta di una rielaborazione dell'articolo apparso anche su Sigplan Notices col titolo 'Reply to a paper by A.N. Habermann on the programming language Pascal'.

● MacLennan B.J., A NOTE ON DYNAMIC ARRAYS IN PASCAL, Sigplan Notices (ACM), vol.10,n.9 (settembre 1975), 39-40.

Prendendo spunto dalle possibilità di definire 'variant records' in Pascal, che l'autore vede come una qualche forma di caratteristica dinamica, viene proposta una estensione ad array dinamici non in contraddizione con gli scopi e con lo spirito di Pascal.

● Wirth N., COMMENT ON A NOTE ON DYNAMIC ARRAYS IN PASCAL, Sigplan

Notices (ACM), vol.11, n.1 (gennaio 1976), 37-38.

Wirth risponde alle proposte di MacLennan proponendo a sua volta delle alternative e criticandone alcune affermazioni.

● Steensgaard-Madsen J., MORE ON DYNAMIC ARRAYS IN PASCAL, Sigplan Notices (ACM), vol.11, n.5 (maggio 1976), 63-63.

Piccola nota nella discussione MacLennan-Wirth.

● Kittlitz E.N., BLOCK STATEMENTS AND SYNONYMS FOR PASCAL, Sigplan Notices (ACM), vol.11, n.10 (ottobre 1976), 32-36.

E' descritto un linguaggio, PYXIS, che è il risultato di modifiche apportate alla versione originaria di Pascal. In PYXIS sono implementate due nuove strutture viste dall'autore come ragionevoli estensioni: sinonimi e block statements. Il linguaggio è operativo su un CDC CYBER 175.

● Pokrowsky S., FORMAL TYPES AND THEIR APPLICATION TO DYNAMIC ARRAYS IN PASCAL, Sigplan Notices (ACM), vol.11, n.10 (ottobre 1976), 36-42.

"The formal type concept is presented as a means to uniformly introduce in the Pascal language the dynamic array facility (which may be done as a pure extension) and formal procedure specifications (which would require some changes in the standard language)". Secondo l'autore, le soluzioni proposte presentano numerosi vantaggi: "[...] they are logical and uniform, consistent with systematic style of programming; they are efficient at both compile and run-time and they considerably enhance the expressive power of the language [...]"

● Conradi R., FURTHER CRITICAL COMMENTS ON PASCAL PARTICULARLY AS A SYSTEMS PROGRAMMING LANGUAGE, Sigplan Notices (ACM), vol.11, n.11 (novembre 1976), 8-25.

L'articolo entra nella controversia tra Habermann e Lecarme con commenti dal punto di vista di un programmatore di sistemi. In particolare vengono avanzate proposte di miglioramento e critiche su: tipi di dati, valori iniziali, operatori booleani, frasi case, for, with, puntatori, vettori, tipi scalari, subranges, insiemi ed array impaccati.

● Kittlitz E.N., ANOTHER PROPOSAL FOR VARIABLE SIZE ARRAYS IN PASCAL, Sigplan Notices (ACM), vol.12, n.1 (gennaio 1977), 82-86.

"The syntax, semantics and some implementation details for a flexible array bound capability in Pascal are discussed. The constructs described are currently implemented as part of the PYXIS system at the University of Calgary. PYXIS is the result of more than two years of modifying Pascal and the compiler provided for it."

● Condict M.N., THE PASCAL ARRAY CONTROVERSY AND A METHOD FOR ENFORCING GLOBAL ASSERTION, Sigplan Notices (ACM), vol.12, n.11 (novembre 1977), 23-28.

Un nuovo articolo nella controversia sulla mancanza di array dinamici in Pascal. Contiene anche una proposta sull'inserimento in Pascal di asserzioni. Con esempi.

● Welsh J., Sneeringer W.J., Hoare C.A.R., AMBIGUITIES AND INSECURITIES IN PASCAL, Software-Practice and Experience, vol.7, n.6 (novembre-dicembre 1977), 685-696.

E' trattato un insieme di insicurezze e di ambiguità nella definizione di Pascal. Le ambiguità riguardano: a) i costruttori d'insieme, b) le scope-rules in relazione alla compilazione, c) l'equivalenza dei tipi con l'analisi delle due possibili definizioni di tipo (equivalenza di nomi ed equivalenza strutturale). Tra

le insicurezze sono analizzati i variant records, procedure e funzioni come parametri, violazioni di range, variabili non inizializzate, accesso di variabili dinamiche.

● Iglewski M., Madey J., Matwin S., A CONTRIBUTION TO AN IMPROVEMENT OF PASCAL, Sigplan Notices (ACM), vol.13, n.1 (gennaio 1978), 48-59.

Si tratta di un'analisi dei punti deboli del linguaggio e della sua descrizione, unita ad alcune proposte di cambiamenti ed eventuali correzioni senza modificare l'essenza del linguaggio. Alcune modifiche contemplate sono: a) espressioni sulla destra di definizioni di costanti, b) l'uso di 'intervalli' di label in frasi case, c) l'inizializzazione di variabili. Sono riviste criticamente anche le affermazioni di Conradi (citato precedentemente).

c) Definizioni formali

● Hoare C.A.R., Wirth N., AN AXIOMATIC DEFINITION OF THE PROGRAMMING LANGUAGE PASCAL, Acta Informatica, vol.2, n.4, 1973, 335-356.

"The programming language Pascal was designed as a general purpose language efficiently implementable on many computers and sufficiently flexible to be able to serve in many areas of application. Its defining report was given in the style of the Algol 60 report. A formalism was used to define the syntax of the language rigorously. But the meaning of programs was verbally described in terms of the meaning of individual syntactic constructs. Its disadvantage is that many semantic aspects of the language remain sufficiently imprecisely defined to give rise to misunderstanding [...] The axiomatic definition method proposed in 'An axiomatic basis for computer programming' is extended and applied to define the meaning of the programming language Pascal. The whole language is covered with the exception of real arithmetic and go to statements [...] "

● Tennent R.D., A DENOTATIONAL DEFINITION OF THE PROGRAMMING LANGUAGE PASCAL, Technical Report 77-47, (luglio 1977), Departement of Computing and Information Science, Queen's University, Kingston, Ontario, Canada, pp.44.

"This report presents a formal definition of the semantics of the programming language Pascal, including static aspects such as scope and type checking, using the concepts and notation of denotational semantics. It is suggested that the definition can be used as a standard from which to derive complete informal description, valid proof rules, and correct implementations."

d) Compilatori ed implementazioni del linguaggio

● Wirth N., THE DESIGN OF A PASCAL COMPILER, Software-Practice and Experience, vol.1, 1971, 309-333.

● Welsh J., Quinn C., A PASCAL COMPILER FOR THE ICL 1900 SERIES COMPUTERS, Software-Practice and Experience, vol.2, n.6 (giugno 1973), 235-244.

● Desjardins P., A PASCAL COMPILER FOR THE XEROX SIGMA 6, Sigplan Notices (ACM), vol.8, n.6 (giugno 1973), 34-35.

E' la descrizione sintetica dello sviluppo di un compilatore Pascal per un Xerox Sigma 6. Non si tratta di un compilatore originale ma di uno derivato da quello implementato sul CDC CYBER 74.

● Ammann U., THE METHOD OF STRUCTURED PROGRAMMING APPLIED TO THE DEVELOPMENT OF A COMPILER, International Computing Symposium 1973, North-Holland Publ. Co., 1974, 93-99.

"This note reports about an attempt to develop a compiler for the ALGOL-like programming language PASCAL by following Dijkstra's method of structured programming. The main development steps are described. Several examples of parallel refinement of data and program structures

are given. Some familiarity with PASCAL or at least the data structures introduced by Hoare is supposed."

● Amble T., Molster T., Sundvor V., UNIT PASCAL SYSTEM FOR THE UNIVAC 1108 COMPUTER, Institut for Databehandling, University of Trondheim, 1974.

● Crespi Reghizzi S., Garbagnati D., Morpurgo R., IL COMPILATORE PASCAL PER L'UNIVAC 1108. Parte I: COMPILAZIONE MEDIANTE UN INTERPRETE SCRITTO IN FORTRAN, Ist. di Elettronica ed Elettrotecnica del Politecnico di Milano, Internal Report 74-26, 1974.

● Feiereisen L., IMPLEMENTATION OF PASCAL ON THE PDP-11/45, DECUS Conference, Zuerich, settembre 1974.

● Nori K. et al., THE PASCAL P-CODE COMPILER: IMPLEMENTATION NOTES, ETH Zuerich Berichtedas fuer Informatik, n.10, 1975.

● Iglewski M., Krepski A., Missala M., A PASCAL COMPILER FOR THE CDC 6000 COMPUTER AND ITS TRANSFERRING TO THE IBM 360/370 COMPUTERS, Elektron. Informationsverarb. Kybern., n.11, 1975, 352-359.

● Grosse Lindeman C.O., Nagel H.H., POSTLUDE TO A PASCAL COMPILER BOOTSTRAP ON A DECSYSTEM 10, Software-Practice and Experience, vol.6, n.1 (gennaio-marzo 1976), 29-42.

L'articolo descrive l'implementazione di un compilatore Pascal derivato dal P-compilatore di Wirth, migliorato nell'efficienza tramite la generazione di codice per il DECSystem 10 invece che per un ipotetico stack computer. Sono state aggiunte anche nuove facilities al linguaggio per adattarlo alle necessità di un ambiente time-sharing. Sono elencati anche i tempi di sviluppo e i risultati quantitativi dell'implementazione.

● Bron C., De Vries W., A PASCAL COMPILER FOR PDP-11 MINICOMPUTER, Software-Practice and Experience, vol.6, n.1 (gennaio-marzo 1976), 109-116.

L'articolo descrive lo sviluppo di un cross-compiler per la famiglia di calcolatori PDP-11. Sono discusse le motivazioni della scelta di Pascal e dei vantaggi insiti nello sviluppo del progetto a partire da un compilatore esistente ed adatto ad essere modificato. Gli autori concludono con una discussione sull'architettura dei PDP-11.

● Stocks A.I., THE PASCAL 11 PROGRAMMING SYSTEM, PhD Thesis, Luglio 1976, University of Illinois, Urbana.

"The initial approach to implementing Pascal on the PDP-11 was to obtain and critically examine the compiler for Pascal written for the CDC 6000 series of machines. This compiler is itself written in Pascal and it was hoped that it could have been modified by rewriting the code generator to produce code for the PDP-11 directly. Unfortunately the compiler contained both overt and covert machine dependencies [...] The approach taken in the bootstrap compiler was to split the compiling activity into two stages. Firstly a Macro-11 program was written, which took a Pascal program, and produced code for an hypothetical Pascal source language machine (SLM). Then the code for the Pascal SLM was translated into Macro-11 code, and the assembler on the PDP-11 then produced the final object code corresponding to the original program [...] "

● Russel D.L., Sue J.Y., IMPLEMENTATION OF A PASCAL COMPILER FOR THE IBM 360, Software-Practice and Experience, vol.6, n.3 (luglio-settembre 1976), 371-376.

E' descritta l'implementazione di un compilatore Pascal, sviluppato a partire dal compilatore originale di Wirth per il CDC 6000 scritto in Pascal. Il risultato è un compilatore scritto in PL/I piuttosto inefficiente, utilizzato per compilare il compilatore di Wirth (modificato

per generare codice IBM 360). Il compilatore di Wirth è stato quindi compilato da sé stesso ottenendo un compilatore efficiente. I passi di bootstrap sono descritti.

● Crespi Reghizzi S., Garbagnati D., Morpurgo R., A PASCAL COMPILER FOR THE UNIVAC MACHINES, Proceedings of AICA Annual Congress, Milano, ottobre 1976.

"The realization of a Pascal compiler for the UNIVAC 1100 machines is described. This was done by transporting the original Pascal compiler for an hypothetical stack machine to the U-1108 with the 'full-bootstrapping' technique, that is all work was done on the target machine".

● Fischer C.N., LeBlanc R.J., EFFICIENT IMPLEMENTATION AND OPTIMIZATION OF RUN-TIME CHECKING IN PASCAL, Proceedings of an ACM Conference on Language Design for Reliable Software, Sigplan Notices (ACM) vol.12, n.3 (marzo 1977), 19-24.

"Complete run-time checking of programs is an essential tool for the development of reliable software. A number of features of the programming language Pascal (arrays, subranges, pointers, record variants, formal procedures)can require some checking at run-time as well as during compilation. The problem of efficiently implementing such checking is considered. Language modification to simplify such checking are suggested. The possibility of optimizing such checking is discussed [...] "

● Garbagnati D., Princi D., Savoretti F., Toscani A., MODIFICHE DEL COMPILATORE PASCAL PER L'UNIVAC 1108, Ist.di Elettronica ed Elettrotecnica del Politecnico di Milano, Internal Report, Giugno 1977.

● Ammann U., ON CODE GENERATION IN A PASCAL COMPILER, Software-Practice and Experience, vol.7, n. 3 (giugno-luglio 1977), 391-424.

"This report deals with code generation in a Pascal compiler. It gives insight into the run-time organisation of data and the use of the hardware registers of the underlying machine (a CDC 6400). It is shown how the compiler maintains a description of the register contents and uses this description to generate efficient code. Several examples of compiled code are discussed."

● Garbagnati D., PASCAL COMPILER, Clup, Milano, 1977, pp.47.

"The Pascal compiler is a program which accepts Pascal statements and produces assembly language programs for the UNIVAC 1100 series system. The compiled program, in order to be executed, has to be assembled by the U-1100 Assembler processor. The compiler itself is written in machine language and is called 'Pascal-Milano Compiler' [...] "
E' la guida per l'uso da parte degli utenti del compilatore.

● Bates D., Cailliau R., EXPERIENCE WITH PASCAL COMPILERS ON MINICOMPUTERS, Sigplan Notices (ACM), vol.12, n.11 (novembre 1977), 10-23.

"This paper relates the history of an implementation of the language Pascal on a mini computer. The unnecessary difficulties encountered on the way, led the authors to reflect on the distribution of 'portable' compilers in general and suggest some guidelines for the future. Their experiences shown that it should be possible to implement a P4 Pascal system on any 16-bit minicomputer in less than two man months, given an implementor already familiar with the target machine."

● Welsh J., ECONOMIC RANGE CHECKS IN PASCAL, Software-Practice and Experience, vol.8, n.1 (gennaio-febbraio 1978), 85-98.

"A Pascal implementation is described

which exploits the information provided by subrange type declarations to minimize the run-time checking involved in detecting range violations. An evaluation of its performances is given and some possible modifications are discussed." Un'analisi particolare è rivolta alle frasi FOR e CASE.

e) Confronti con altri linguaggi

● Venema T., Des Rivieres J., EUCLID AND PASCAL, Sigplan Notices (ACM), vol. 13, n.3 (marzo 1978),57-69.

L'articolo esamina le differenze tra Euclid e Pascal. Queste differenze sono raggruppate in tre grandi classi: a) cambiamenti riguardanti l'utente (strutture di dati), b) programmazione di sistemi (allocazione dinamica etc.), c) problemi di verificabilità.

f) Introduzioni alla programmazione in Pascal

● Conway R., Gries D.,Zimmermann E.C., A PRIMER ON PASCAL,Winthrop Publishers Inc., Cambridge - Massachusetts, 1976, pp. 433

Un libro di introduzione alla programmazione strutturata con l'uso di Pascal.

● Webster C.A.G., INTRODUCTION TO PASCAL, Heyden & Son Ltd, 1976, pp. xi-129.

"This book is particularly intended to help those taking a first course in computer programming using the programming language Pascal as a vehicle. The coding examples given therefore aim to provide a variety of straightforward models on which to build as the reader's skill with Pascal grows, rather than be pieces of programming tricks. [...] The approach adopted was to describe in the body of the book an essentially full version of Pascal, discussing where appropriate the alternative philosophies of the language as originally devised and revised, and relegate to appendices the minutiae of character sets and the like used in the principal implementation."

g) Standard di programmazione in Pascal

● Hueras J., Ledgard H., AN AUTOMATIC FORMATTING PROGRAM FOR PASCAL, Sigplan Notices (ACM),vol.12, n.7 (luglio 1977), 82-84.

Si tratta della descrizione di un programma altamente modulare interamente scritto in Pascal e implementato su un CDC CYBER 74, che realizza una corretta impaginazione di programmi Pascal, a tutto vantaggio della leggibilità del programma, ottenuta senza imporre restrizioni o condizioni al programmatore. Una copia del programma è ottenibile rivolgendosi a Ledgard.

● Ledgard H., Singer A., Hueras J., A BASIS FOR EXECUTING PASCAL PROGRAMMERS, Sigplan Notices (ACM), vol.12, n.7 (luglio 1977), 101-105.

"Any programmer who fails to comply with the standard naming, formatting or commenting convention should be shot" (M.Spier, D.E.C.). Viene presentato un insieme di regole standard cui i programmatori Pascal dovrebbero attenersi. Queste regole riguardano la documentazione dei programmi, l'uso delle strutture di controllo, i formati di impaginazione etc.

● J.L. Peterson, ON THE FORMATTING OF PASCAL PROGRAMS, Sigplan Notices (ACM),vol.12, n.12 (dicembre 1977), 83-86.

Un altro articolo sull'impaginazione di programmi Pascal. Sono descritti formati standard per ogni frase Pascal.

● Bates D., Letter to the Editor, SIGPLAN Notices, vol.13, n.3 (marzo 1978), 12-15.

Alcuni commenti sul lavoro di Peterson, 'On the formatting of Pascal programs'.

h) Pascal e la didattica

● Lecarme O., STRUCTURED PROGRAMMING, PROGRAMMING TEACHING AND THE LANGUAGE PASCAL, Sigplan Notices (ACM), vol.9, n.7 (luglio 1974), 15-21.

"This paper is a series of considerations on the three subjects stated in its title, and on their mutual relationship. Since it is a survey of the present situation in three related and evolving areas, references play a very important role [...] "

● Nutt G.J., A COMPARISON OF PASCAL AND FORTRAN AS INTRODUCTORY PROGRAMMING LANGUAGES, Sigplan Notices (ACM), vol.13, n.2, (febbraio 1978), 57-62.

Vengono confrontati Pascal e Fortran come linguaggi di introduzione alla programmazione. Vengono fornite delle statistiche basate su corsi tenuti all'Università del Colorado: esse riguardano le difficoltà di apprendimento dei due linguaggi e dei rispettivi tempi di apprendimento di uno, una volta noto l'altro.

i) Applicazioni e strumenti

● Marmier E., A PROGRAM VERIFIER FOR PASCAL, Information Processing 1974 (IFIP Congress 1974), North-Holland, 1974, 177-181.

"The paper presents current work on a program verifier for PASCAL programs. A list of requirements which any useful program verification system should meet is suggested. Some notational facilities meant to be supported by the author's program verifier are discussed, including (recursive) functions and predicates, a restricted use of quantifiers, the bounded μ-operator, non-program variables, and a notation for a certain class of sets of pointers. Finally, an example of a possible input to the program verifier is shown".
L'autore appartiene al gruppo di Wirth.

● Matwin S., Missala M., A SIMPLE MACHINE INDEPENDENT TOOL FOR OBTAINING ROUGH MEASURES OF PASCAL PROGRAMS, Sigplan Notices (ACM), vol.11, n.8 (agosto 1976), 42-44.

"The goal of this project was to obtain support for measuring very large Pascal programs. [...] The main problem was the proportion between the amount of time necessary for construction of such a tool and the quantity of information provided by the use of the final product on large programs. [...] Since procedure is the basic entity of any Pascal program we decided to obtain statistics of the relative amount of time spent in each procedure as the output of our system. The whole system is written in Pascal [...] "

● Biedl A., AN EXTENTION OF PROGRAMMING LANGUAGES FOR NUMERICAL COMPUTATION IN SCIENCE AND ENGINEERING WITH SPECIAL REFERENCE TO PASCAL, Sigplan Notices (ACM), vol.12, n.4 (aprile 1977), 31-33.

Partendo dai requisiti richiesti ad un linguaggio per l'analisi numerica, viene sviluppata un'analisi di Pascal ed in particolare della sua gestione dei tipi REAL e della sua aritmetica reale.

● Mohilner P.R., USING PASCAL IN A FORTRAN ENVIRONMENT, Software-Practice and Experience, vol.7, n.3 (giugno-luglio 1977), 357-362.

"The ability of use existing libraries is essential if a programming language is to be of practical value. This paper presents the experience gained in using Pascal in a Fortran environment and details the resolution of problems encountered in file communication, parameter passing and establishing default values under the condition of mixed language programming [...]"

AVVENIMENTI

SUCCESSFUL SOFTWARE MANAGEMENT TECHNIQUES CONFERENCE

Organizzato dalla Honeywell, si è tenuta a Minneapolis il 28, 29 e 30 marzo 1978 una conferenza dedicata ai vari aspetti di management di progetti software.

La conferenza, concepita come una naturale continuazione del simposio sulla produttività del software svoltosi sempre a Minneapolis l'anno precedente (vedi NOTE DI SOFTWARE n. 2), aveva lo scopo di stimolare uno scambio di esperienze fra i manager dei gruppi di ingegneria di software di tutte le consociate della Honeywell, in ambedue le branche della Società: Honeywell Information Systems e Honeywell Control Systems.

La conferenza, che ha visto una partecipazione assai attiva di tutte le consociate della Honeywell delle varie parti del mondo e di alcuni utenti e società di consulenza, testimonia l'importanza crescente attribuita alle tecniche di management, considerate come uno dei fattori determinanti per una buona produttività.

Obiettivo della conferenza era uno scambio di esperienze, sia su progetti software di grande dimensione per sistemi 'general-purpose', che su progetti di piccola dimensione oltre che per i cosiddetti 'embedded computer systems' (sistemi logicamente incorporati in un sistema più grande - un'apparecchiatura elettromeccanica, una nave, un aereo, un sistema di comunicazione - la cui funzione primaria non è il calcolo).

I contributi dei partecipanti (31 presentazioni distribuite in 11 sessioni) hanno trattato i seguenti argomenti:

- organizzazione dei gruppi e management dell'attività
- tecniche di pianificazione, di misura, di 'reporting'
- 'guidelines' e 'policies' per lo sviluppo del software
- controllo del disegno
- la "fabbrica del software"
- tecniche di documentazione
- metodi di collaudo e supervisione della qualità.

Le presentazioni, tutte derivate da esperienze reali accompagnate da successo (in alcuni casi dopo l'adozione di adeguati provvedimenti correttivi, di cui sono stati descritti i benefici), hanno messo in evidenza una significativa comunanza di problemi, e una buona convergenza nelle soluzioni adottate.

Presentiamo, qui di seguito, una sintesi dei vari aspetti trattati.

Organizzazione dei gruppi e management dell'attività

E' stata da tutti ricordata la fondamentale importanza di una definizione completa e precisa del processo di produzione, organizzata in fasi distinte, con input, output e responsabilità chiaramente definiti, e in modo tale che la sua completezza sia verificabile.

E' peraltro emerso che la struttura organizzativa dei gruppi si deve adeguare al modo di lavorare. I gruppi di supporto o non direttamente produttivi (System Engineering, integrazione, controllo qualità, pianificazione, gestione del centro di calcolo) sono frequentemente separati dai gruppi di sviluppo; ove le caratteristiche del progetto lo suggeriscano, viene tuttavia a-

dottata, in alternativa, una struttura organizzativa "a matrice", coordinata da un project manager.

E' stato rilevato il ruolo determinante di un buon management nel favorire una comunicazione efficiente fra i gruppi - considerata una condizione essenziale per la partecipazione di tutti agli aspetti globali del progetto - e nell'incoraggiare le innovazioni, sia tecnologiche (ad esempio, programmazione strutturata, strumenti per una migliore automazione, text editor), che di metodo (uso di standard comuni, misure, crescita professionale dei collaboratori, ecc.).

Tecniche di pianificazione, di misura, di reporting

Fra le cause maggiori di insuccesso è stata individuata la sottovalutazione della funzione di pianificazione, che ha come conseguenza stime non corrette. Molte presentazioni hanno infatti posto l'enfasi sulla necessità di una pianificazione completa di tutto il progetto, che stimi tempi e costi (uomo e macchina) di ogni fase con adeguato anticipo.

Connessa all'attività di pianificazione, è da segnalare la pratica di 'review' sia esterne (cioè con gli utenti o loro rappresentanti) che interne (sul disegno e la qualità), come lo strumento più efficace per un controllo "lungo la strada" del progetto stesso.
Considerate molto produttive le 'review' di tipo 'walkthrough', che coinvolgono direttamente i partecipanti al progetto.

Grande importanza è stata inoltre attribuita da tutti ai meccanismi di 'change control'.

'Guidelines' e 'policies' per lo sviluppo del software

L'applicazione di precise linee guida che stabiliscano:

- quali documenti produrre (piani, specifiche, ...)
- quali attività effettuare
- quali 'review' sostenere

nelle varie fasi di un progetto è considerata la via migliore per ottenere una buona visibilità del progetto stesso (da parte di tutti i partecipanti, e non solo da parte del management), per un controllo effettivo delle aree di rischio, per favorire comunicazioni chiare, per unificare metodi e strumenti.

Tra le presentazioni che hanno trattato questi argomenti, è da segnalare quella di R. D. Krutz (direttore del Corporate Computer Science Center della Honeywell) il quale, presentando come esempio di riferimento rilevante le 'policies' adottate dalla società americana TRW, ha affermato che "l'applicazione sistematica di 'policies' e 'guidelines' è la via più economica per un trasferimento rapido di nuove tecnologie e per una comprensione di esse". J.S. Gansler, inoltre, ha descritto il grande sforzo che il Department of Defense (DoD) americano sta compiendo in questo settore. Le direttive emesse dal DoD, che può considerarsi il maggiore utilizzatore di elaboratori (circa 3 miliardi di dollari di spesa annua, solo per il software), impongono ai produttori di software-sia industrie che case di consulenza-dei precisi requisiti di qualità, di metodo e di controllo del progetto, al fine di ridurre i costi complessivi del software prodotto, sul suo intero arco di vita.
Tali direttive corrispondono a un grosso e recente sforzo culturale che, inevitabilmente, produrrà effetti benefici sulla produttività e professionalità dei produttori e degli utenti (instaurando delle 'policies' per l'accettazione del prodotto).

Sempre in questo ambito di standardizzazione e unificazione, è da riferire l'attività del Department of Defense per arrivare a unificare i linguaggi di programmazione di alto livello usati al suo interno (oggi il DoD usa 450 linguaggi o dialetti differenti, su 200 modelli diversi di calcolatori). Un recente articolo sulla rivista Computer [1] riassume la storia di questo sforzo volto alla definizione di un linguaggio di alto livello, adatto alla produzione di software che soddisfi le esigenze di utenti sia di sistemi 'general-purpose' che di sistemi speciali.

Gli obiettivi definiti per il nuovo linguaggio sono stati tradotti in requirement dettagliati [2], a fronte dei quali il DoD ha effettuato una selezione di 15 proposte di disegno, provenienti da varie case produttrici di software [1].

S. Vestal, del System and Research Center della Honeywell, ha esposto nella conferenza il lavoro fatto dalla Honeywell in relazione al DoD Common Programming Language, e ha riferito come la Honeywell sia stata selezionata dal DoD tra i 15 concorrenti, insieme ad una seconda società, per proseguire oltre la fase di disegno e realizzare il compilatore del linguaggio sul quale compiere verifiche di uso effettivo.

Controllo del disegno

In diverse presentazioni, sono emersi due punti fondamentali:

- necessità di un coinvolgimento attivo e rilevante dell'utente nella definizione dei requisiti e nella revisione formale delle specifiche, per garantirne la completezza e minimizzare così i rischi di modifiche successive;
- adozione sistematica di review interne di disegno (affidate a sistemisti molto competenti, dedicati talvolta a questa attività), con lo scopo di verificarne la correttezza e la completezza prima di iniziarne l'implementazione.

La fabbrica del software

L'importanza di una buona "fabbrica del software" è stata sottolineata in tutte le presentazioni. In particolare, valutazioni assai positive sono state espresse sull'uso di text-editor per produrre documentazione, e di terminali interattivi per facilitare l'accesso al sistema e lo sviluppo dei programmi e delle prove.

Rilevante l'enfasi posta sui seguenti aspetti, indicati come essenziali per una buona produttività nelle fasi implementative:

- uso di linguaggi di programmazione di alto livello
- impiego di tecniche di programmazione strutturata
- strumenti per la gestione di librerie e per la costruzione automatica di sistemi
- simulatori
- supporto efficiente al centro di calcolo.

Quattro presentazioni sono state dedicate al MULTICS (il più grosso sistema oggi costruito dalla Honeywell), delineandone nei loro vari aspetti le sue eccezionali capacità come fabbrica di software.
Del MULTICS si è soprattutto rilevato un insieme di caratteristiche ineguagliate sul mercato:

- facilità d'uso per l' 'end-user'
- strumenti per la gestione e l'aggiornamento dei programmi e delle librerie di progetto
- strumenti (text-editor, word-processing) assai efficienti per la documentazione, figure incluse
- capacità di supportare in modo automatico l'organizzazione di un progetto e la comunicazione tra i componenti di un progetto e tra progetti
- privacy e security di assoluto affidamento, a protezione delle librerie di progetto.

Un utente privato ha presentato il suo uso di MULTICS come fabbrica del software distribuita, collegata ad alcuni sistemi HoneywellLevel 6 come satelliti. Lo stesso concetto di fabbrica del software distribuita è stato adottato dalla Honeywell Control Systems, che a partire da quest'anno lo utilizzerà per la produzione di software per calcolatori speciali e per microprocessor.

Tecniche di documentazione

Quattro presentazioni erano dedicate alla documentazione. La scelta di tecnologie appropriate è stata dichiarata da tutti essere un elemento determinante per la esistenza stessa della documentazione: essa, senza la disponibilità di opportuni strumenti automatici, spesso non viene aggiornata. Il controllo, la manutenzione, la continuazione, le review del progetto non si possono effettuare - è stato affermato - senza documentazione; d'altra parte, una buona documentazione di disegno riduce il numero degli errori prodotti.

Oltre all'enfasi posta sull'uso di text-editor da terminale interattivo, dalle presentazioni è apparso evidente che la tecnica di inserire la documentazione dettagliata di disegno, mediante appositi commenti, direttamente nel codice del programma tenderà a sostituire l'uso dei flowchart.

Si veda nel seguito del presente articolo per la presentazione della HISItalia sulla metodologia di documentazione PSPN.

Metodi di collaudo e supervisione della qualità

In quest'area sono da rilevare, oltre all'importanza attribuita all'attività di testing e ai gruppi a ciò dedicati, i seguenti punti, indicati nelle presentazioni come fondamentali nel management di progetto:

- pianificazione tempestiva (al termine della fase di disegno) delle attività di testing
- definizione di 'release criteria', numerici e qualitativi, la cui verifica costituisca uno strumento di supporto alle decisioni intermedie e finali
- esposizione del software prodotto a un uso reale (presso clienti pilota, presso centri EDP interni, ...) prima del rilascio
- strumenti per testing automatico e per misura di prestazioni.

Partecipazione della HISI

L'Ingegneria della HISI ha partecipato con le seguenti presentazioni:

- A. Cicu, "HIS Italia GCOS Software Engineering Organization"
- P. Ghelfi, "Planning, Supporting, Controlling the Integration Activity of HIS Italia GCOS 62 Software Releases"
- M. Rivolta, "GCOS 62 Qualification Approach"
- A. Cicu, G. Degli Antoni, M. Maiocchi, R. Polillo, G. Torriani, "An Integrated Multilevel Documentation Approach".

Le prime tre presentazioni hanno trattato i principi organizzativi e di metodo adottati dall'Ingegneria HISI per le attività di pianificazione, integrazione e collaudo. La quarta ha esposto la metodologia di documentazione PSPN (cfr. NOTE DI SOFTWARE n. 3), che ha raccolto commenti assai favorevoli, per la efficacia didattica e di comunicazione, e per la ottimizzazione dei contenuti del manuale utente derivata dai criteri di produzione indicati dalla metodologia stessa.

Due presentazioni, infine, nonchè l'introduzione alla conferenza, hanno trattato esperienze inerenti le misure di qualità, quantità e costi, sottolineando l'importanza della loro accuratezza in fase previsionale e consuntiva.

La relazione del vicepresident della Honeywell, J. J. Renier, ha ben sintetizzato l'esigenza di una accurata attenzione alla componente economica dei progetti, con l'invito ai manager software a trasformarsi da "innovatori a innovatori con spirito imprenditoriale".

A. Cicu, P. Ghelfi

[1] D.A. Fisher - DoD's Common Programming Language Effort, Computer, Marzo 1978

[2] Department of Defense Requirements for High Order Computer Programming Languages, Revised "IRONMAN", The Technical Requirements, SIGPLAN Notices, Vol. 12, n. 12 (dicembre 1977)

IL SEGNALIBRO

In questa rubrica vengono segnalati libri, articoli o studi di recente pubblicazione che, a giudizio della redazione o dei lettori di NOTE DI SOFTWARE, rivestono un particolare interesse nell'ambito dei temi a cui questa rivista è dedicata.

Le citazioni fra virgolette, se non ne è indicata la fonte, sono tratte dai lavori stessi.

Project Management

● Ingrassia, F.S., COMBATING THE "90% COMPLETE" SYNDROME, Datamation, vol.24, n.1 (gennaio 1978), 171-176.

"One of the most common problems encountered in software management is the lack of visibility into the progress being made by each individual on a project. Consider the "percent complete" estimates given by programmers. They are notoriously bad. I have had cases of a programmer, whose job took twelve full-time weeks to complete, reporting "90% complete" by the end of the fifth week. [....] One way you can combat this "90% complete" syndrome is to establish mini-milestones for each programmer's progress. Even with this plan, though, you can run into problems if the milestones aren't easily and objectively verifiable. [....] Thus, you need something better: a way of coupling the milestones, the artifacts they represent, their schedule, and an objective scheme for verifying they have been completed. At TRW this need is being met by a management tool called a Unit Development Folder, or UDF. [....] The UDF is, simply, a particular form of development notebook: a three ring binder containing a cover sheet and several predefined common sections. This notebook has proven useful in collecting and organizing components of software products as they are produced. In essence, however, the UDF is much more than that: it is a means of imposing a management philosophy and a development methodology on an activity that often appears chaotic. The UDF provides a uniform and visible collection point for all documentation and code associated with each unit (a "unit" consists of a routine or group of related routines, typically comprising about 50 to 300 source statements). [....] "

Linguaggi di specifica

● Balzer, R., Goldman, N., Wile, D., INFORMALITY IN PROGRAM SPECIFICATIONS, IEEE Transactions on Software Engineering, vol.SE-4, n.2 (marzo 1978), 94-103.

"This paper is concerned with the need for computer-based tools which help human designers formulate formal process-oriented specifications. It first determines some attributes of a suitable process-oriented specification language, then examines the reasons why specifications would still be difficult to write in such a language in the absence of formulation tools. The key to overcoming these difficulties seems to be the careful introduction of informality based on partial, rather than complete, descriptions and the use of a computer-based tool that uses context extensively to complete these descriptions during the process of constructing a well-formed specification. Some results obtained by a running prototype of such a computer-based tool on a few informal example specifications are presented and, finally, some of the techniques used by this prototype system are discussed."

Linguaggi di programmazione

● Wirth, N., PROGRAMMING LANGUAGES: WHAT TO DEMAND AND HOW TO

ASSESS THEM, in SOFTWARE ENGINEERING, edited by R.H. Perrott (Proceedings of a Symposium held at The Queen's University of Belfast, 1976), Academic Press (A.P.I.C. Studies in Data Processing n.14), 1977, pagg.155-173.

"The software inflation has led to a software crisis which has stimulated a search for better methods and tools. This includes the design of adequate system development languages. This paper contains some hints on how such languages should be designed and proposes some criteria for judging them. It also contains suggestions for evaluating their implementations, and emphasizes that a clear distinction must be made between a language and its implementation. The paper ends with concrete figures about a Pascal implementation that may be used as yardstick for objective evaluations."

● Richard, F., Ledgard, H.F., A REMINDER FOR LANGUAGE DESIGNERS, SIGPLAN Notices (ACM), vol.12, n.12 (dicembre 1977), 73-82.

"Most existing programming languages do not suit the production of large software systems. To implement real problems, no current programming language offers clear solutions. Too often, the structure of the problem must be twisted to the structure of the language. To devise programs that can be read and modified, programmers must observe a strict obedience to numerous and complex programming standards. Many of these standards exists only to remedy serious pitfalls of the language, and complicate the task of programming. As to verification of large programs, most languages are hopeless. [....] In this paper we suggest principles for UTOPIA 84 (Knuth 74). These principles are based in part on the works of Dijkstra, Gannon and Horning, Hoare, Knuth, Ledgard and Marcotty, Weinberg, Wirth, and Wulf and Shaw. No attempt is made to address the whole language design area. Little consideration is given to writability and efficiency of implementation. We believe that these goals have received too much attention in the past. [...] "

● Fisher, D.A., DoD's COMMON PROGRAMMING LANGUAGE EFFORT, Computer, vol.11, n.3 (marzo 1978), 24-33.

"DoD's (USA Department of Defense) common programming language effort is aimed at reducing the development and maintenance cost and improving the quality of software for embedded computer systems. Here is a brief review of the background, scope, goals, and methods of that effort." " [...] The term 'embedded computer system' was first used in 1974 to denote one that is logically incorporated in a larger system, a ship, an aircraft, or a communications system whose primary function is not computation. Included in the concept of embedded computer systems is the support software necessary to design, develop, and maintain them. Computers used primarily for data processing, scientific, or research applications are not normally included in the embedded computer systems category. [...] " Informazioni analoghe sugli scopi, la storia e lo stato del progetto del DoD sono fornite in:

● Whitaker, W.A., THE U.S. DEPARTMENT OF DEFENSE COMMON HIGH ORDER LANGUAGE EFFORT, SIGPLAN Notices (ACM), vol.13, n.2 (febbraio 1978), 19-29.

● (non firmato), DEPARTMENT OF DEFENSE REQUIREMENTS FOR HIGH ORDER COMPUTER PROGRAMMING LANGUAGES - Revised IRONMAN (July 1977) - THE TECHNICAL REQUIREMENTS, SIGPLAN Notices, vol.12, n.12 (dicembre 1977), 39-54.

"The technical requirements for a common DoD high order programming language given here are a synthesis of the requirements submitted by the Military Departments. They specify a set of language characteristics that are appropriate for embedded computer applications. [...]"

● Il numero di marzo 1978 di SIGPLAN Notices (ACM) (vol.13, n.3) contiene una serie di articoli sul linguaggio di program

mazione Euclid, di ricercatori dell'Università di Toronto, dove il linguaggio viene sviluppato: Chang, E., Kaden, N.E., Elliott, W.D., ABSTRACT DATA TYPES IN EUCLID (pp.34-42); Aseltine, E.G., Spencer, H., ISOLATION OF MACHINE DEPENDENCIES IN EUCLID (pp.43-48); des Rivieres, J., Spencer, H., READABILITY AND WRITABILITY IN EUCLID (pp.49-56); Venema, T., des Rivieres, J., EUCLID AND PASCAL (pp.57-69); Barnard, D.T., Elliott, W.D., Thompson, D.H., EUCLID AND MODULA (pp.70-84); Elliott, W.D., INDEX TO THE EUCLID REPORT (pp.85-89).

● Welsh, J., Sneeringer, W.J., Hoare, C.A.R., AMBIGUITIES AND INSECURITIES IN PASCAL, Software - Practice and Experience, vol.7, n.6 (novembre-dicembre 1977), 685-696.

" [...] At the time that Pascal was first designed and developed, the most fashionable languages in the learned and practical world were ALGOL 68 and PL/I. The discovery that the advantages of a high-level language could be combined with high efficiency in such a simple and elegant manner as in Pascal was a revelation that deserves the title of breakthrough. Because of the very success of Pascal, which greatly exceeded the expectations of its author, the standards by which we judge such languages have also risen. It is grossly unfair to judge an engineering project by standards which have been proven attainable only by the success of the project itself, but in the interests of progress, such criticism must be made. Of the criticisms made in this paper, some identify shortcomings of Pascal which can readily be made good by minor changes to the language or its definition. Others indicate problems for which there is no easy solution whithin the current framework. [...] "

Strutture di controllo

● Pratt, T.W., CONTROL COMPUTATIONS AND THE DESIGN OF LOOP CONTROL STRUCTURES, IEEE Transactions on Software Engineering, vol. SE-4, n.2 (marzo 1978), 81-89.

"The 'control computation' for a loop in a program is that part of the program concerned with the initialization, incrementation, and testing of the variables that determine the flow of control into, through, and out of the loop. The elements of loop control computations are identified and their role in structuring our understanding of loops is analyzed. It is argued, through examples drawn from a Pascal compiler, that the intelligibility of a loop is closely tied to the accessibility and intelligibility of the loop control computation. It is further argued, from an analysis of all the loops in this compiler, that most loop control computations fall in a few standard patterns, mostly concerned with the sequential processing of elements of data structures. In light of these results, common loop control statements are critiqued. It appears that better loop control structures that the 'while', 'repeat-until', and similar statement structures are possible and desirable, and some proposals for better structures are given."

Correttezza dei programmi

● Manna, Z., Waldinger, R., IS "SOMETIME" SOMETIMES BETTER THAN "ALWAYS"? - INTERMITTENT ASSERTIONS IN PROVING PROGRAM CORRECTNESS, Communications of the ACM, vol. 21, n. 2 (febbraio 1978), 159-172.

"This paper explores a technique for proving the correctness and termination of programs simultaneously. This approach, the intermittent-assertion method, involves documenting the program with assertions that must be true at some time when control passes through the corresponding point, but that need not be true every time. The method, introduced by Burstall, promises to provide a valuable complement to the more conventional methods. The intermittent assertion method is presented with a number of examples of correctness and termination proofs. Some of these proofs are markedly simpler than their conventional counterparts. On the other hand, it is shown that a proof of

correctness or termination by any of the conventional techniques can be rephrased directly as a proof using intermittent assertions. Finally, it is shown how the intermittent-assertion method can be applied to prove the validity of program transformations and the correctness of continuously operating programs."

Documentazione

● Grouse, P., "FLOWBLOCKS"-A TECHNIQUE FOR STRUCTURED PROGRAMMING, SIGPLAN Notices (ACM), vol. 13, n. 2 (febbraio 1978), 46-56.

Un altro lavoro sugli "structured flowcharts", proposti inizialmente da Nassi e Shneiderman qualche anno fa.

Software Science

● Fitzsimmons, A., Love, T., A REVIEW AND EVALUATION OF SOFTWARE SCIENCE, ACM Computing Surveys, vol.10, n.1 (marzo 1978), 3-18.

"During recent years, there have been many attempts to define and measure the 'complexity' of a computer program. Maurice Halstead has developed a theory that give objective measures of software complexity. Various studies and experiments have shown that the theory's predictions of the number of bugs in programs and of the time required to implement a program are amazingly accurate. It is a promising theory worthy of much more probing scientific investigation. This paper reviews the theory, called 'software science', and the evidence supporting it. A brief description of a related theory, called 'software physics', is included."

Architettura degli elaboratori

● Tanenbaum, A.S., IMPLICATIONS OF STRUCTURED PROGRAMMING FOR MACHINE ARCHITECTURE, Communications of the ACM, Vol.21, n.3 (marzo 1978), 237-246.

"Based on an empirical study of more than 10,000 lines of program text written in a GOTO-less language, a machine architecture specifically designed for structured programs is proposed. Since assignment, CALL, RETURN, and IF statements together account for 93 percent of all executable statements, special care is given to ensure that these statements can be implemented efficiently. A highly compact instruction encoding scheme is presented, which can reduce program size by a factor of 3. Unlike a Huffman code, which utilizes variable length fields, this method uses only fixed length (1-byte) opcode and address fields. The most frequent instructions consist of a single 1-byte field. As a consequence, instruction decoding time is minimized, and the machine is efficient with respect to both space and time."

● Il numero di gennaio 1978 di Communications of the ACM (vol.21, n.1) è dedicato all'architettura degli elaboratori. Contiene sei lavori, specificamente scritti per questo numero speciale: Lavington, S.H., THE MANCHESTER MARK I AND ATLAS: A HISTORICAL PERSPECTIVE, pagg.4-12; Ibbett, R.N., Capon, P.C., THE DEVELOPMENT OF THE MU5 COMPUTER SYSTEM, 13-24; Borgerson, B.R., Hanson, M.L., Hartley, P.A., THE EVOLUTION OF THE SPERRY UNIVAC 1100 SERIES: A HISTORY, ANALYSIS, AND PROJECTION, 25-43; Bell, C.G. Kotok, A., Hastings, T.N., Hill, R., THE EVOLUTION OF THE DECsystem 10, 44-63; Russell, R.M., THE CRAY-1 COMPUTER SYSTEM, 63-72; Case, R.P., Padegs, A., ARCHITECTURE OF THE IBM SYSTEM/370, 73-96.

"The six papers that appear here cover an interesting span of computer system development. The first two articles in this special issue deal with the machines that have been designed and built at Manchester University, Manchester, England. These Manchester machines

have had a strong influence on the English computer industry and it is interesting to see a discussion of their organization and evolution. The papers on the Univac 1100 series, the DECsystem 10, and the IBM System/370 discuss three of the more successful commercial computer series. Unlike most articles on computer systems, these articles do not describe newly announced machines. All three computer architectures were designed and originally implemented about fifteen years ago. What these articles do explore, which is so rarely done, is to critically examine how and why these computer architectures evolved as they have. [...] Another article is on the CRAY-1 computer. This machine is a very high speed (up to 138 million floating-point operations per second), vector-oriented processor. Unlike the aforementioned articles, this one does not describe a long series of compatible machines. Rather, it discusses an attempt to push technology and architecture to the limits in order to obtain as high a performance as possible. [...] "
(dalla 'Foreword to the Special Issue on Computer Architecture', G.Bell, S.H. Fuller, D.P. Siewiorek, editors).

● Bell, C.G., WHAT HAVE WE LEARNED FROM THE PDP-11?, in PERSPECTIVES ON COMPUTER SCIENCE, From the 10th Anniversary Symposium at the Computer Science Department, Carnegie-Mellon University, edited by A.K. Jones, Academic Press, ACM Monograph Series (1977), 7-38.

"In the six years that the PDP-11 has been on the market, more than 20.000 units in 10 different models have been sold. Although one of the original system design goals was a broad range of models, the actual range of 500 to 1 (in cost and memory size) has exceeded the design goals. The PDP-11 was designed to be a small computer, yet its design has been successfully extended to high-performance models. This paper recollects the experience of designing the PDP-11, commenting on its success from the point of view of its goals, its use of technology, and on the people who designed, built and marketed it."

Ingegneria del software - articoli vari

● Sandewall, E., PROGRAMMING IN AN INTERACTIVE ENVIRONMENT: THE "LISP" EXPERIENCE, ACM Computing Surveys, vol.10, n.1 (marzo 1978), 35-71.

"LISP systems have been used for highly interactive programming for more than a decade. During that time, special properties of the LISP language (such as program/data equivalence) have enabled a certain style of interactive programming to develop, characterized by powerful interactive support for the programmer, nonstandard program structures, and nonstandard program development methods. The paper summarizes the LISP style of interactive programming for readers outside the LISP community, describes those properties of LISP systems that were essential for the development of this style, and discusses some current and not yet resolved issues."

● Chevance, R.J., Heidet, T., STATIC PROFILE AND DYNAMIC BEHAVIOR OF COBOL PROGRAMS, SIGPLAN Notices (ACM), vol.13, n.4 (aprile 1978), 44-57.

"The study related hereafter is part of a more general work concerning the design of high level language oriented machines. Design of such architectures is partly based upon characteristics - of representative programs - at the source language level. The lack of informations about characteristics of COBOL programs has lead us to develop a measurement system. Two kinds of program measurements may be made:
- static, i.e. recording occurrences of the various forms of source language constructions as they appear within user's program, - dynamic, i.e. recording occurrence of the above constructions as they are encountered during program execution [...] "

Il lavoro presenta dati (statici e dinamici) relativi a 50 programmi applicativi Cobol ANS 68 (es.: applicazioni di contabilità generale). Cfr. anche il lavoro di Tanenbaum, citato in questa rubrica.

● Il numero di ottobre-dicembre 1977 della Rivista di Informatica (vol.VII,n.4) raccoglie alcuni dei lavori presentati alle Giornate di Studio Honeywell su "Programmazione strutturata: esperienze e orientamenti", Milano, giugno 1976.

I lavori sono : Ancillotti, P., Boari,M., Lijtmaer, N., METODI PER LA SPECIFICA DEL COORDINAMENTO DEI PROCESSI CONCORRENTI (223-238); Levi, G., Sirovich, F., SVILUPPO DI PROGRAMMI A LIVELLI: ASTRAZIONI, SPECIFICHE, ED ESECUZIONE SIMBOLICA (239-253); Della Vigna,P.,Ghezzi, C., Martelli, A., Sirovich, F.,STRUTTURE DI DATI: LIVELLI DI ASTRAZIONE E METODI DI DEFINIZIONE COME STRUMENTO DI PROGETTO DEI PROGRAMMI (255-270); Degli Antoni, G., Gobbi, G., ASPETTI RELATIVI ALLA DOCUMENTAZIONE DEI PROGRAMMI (271-284); Maiocchi, M., IL TRATTAMENTO DELLE CONDIZIONI DI ERRORE NELLA PROGRAMMAZIONE STRUTTURATA (285-293); Scrignaro, D., Taroni, P., Tucci, T., LE LIBRERIE DI PROGETTO COME STRUMENTO DI CONTROLLO DELLA PRODUZIONE SOFTWARE (295-301); Batini, C., D'Atri, A., PROGETTO TOP-DOWN DI BASI DI DATI NEL MODELLO RELAZIONALE (303-319); Boari, M.,Grazia, G., PROGETTO E REALIZZAZIONE DEL SOFTWARE PER UNA CENTRALE DI COMMUTAZIONE TELEFONICA (321-332).

● Anderson, T., Shrivastava, S.K., RELIABLE SOFTWARE: A SELECTIVE ANNOTATED BIBLIOGRAPHY, Software - Practice and Experience, vol.8, n.1 (gennaio-febbraio 1978), 59-76

"A total of 64 references topapers, books and conference proceedings on the subject of software reliability have been selected. Each of these references is provided with an annotation consisting of a paragraph of commentary. Sections of the bibliography are devoted to requirements definition, programming methodology, certification, fault-tolerance and reliability modelling."

● De Michelis, G., Lanzarone, G.A., Simone, C., Vianello, P., 400 VOCI DI SOFTWARE, Supplemento alla rivista "Sistemi e Automazione" n. 181 (aprile 1978), pp. 30.

Un breve dizionario di 400 termini inglesi, scelti fra quelli più in uso nel mondo del software.

Text editors

● Macleod, I.A., DESIGN AND IMPLEMENTATION OF A DISPLAY ORIENTED TEXT EDITOR, Software - Practice and Experience, vol.7, n.6 (novembre-dicembre 1977), 771-778.

"This paper outlines the design and implementation of a text editor which is designed for use on an alphanumeric display terminal with cursor control. Text may be edited directly on the screen by deleting, overstriking or inserting characters. There also exist commands to manipulate and search sections of text. A major design consideration was that the editor should be easily usable by untrained casual users."

Un libro di Kernighan e Plauger

● Kernighan, B.W., Plauger, P.J., SOFTWARE TOOLS, Addison-Wesley Publishing Company, 1976, pagg.338.

Segnaliamo questo libro,anche se non recentissimo, per il suo notevole interesse. Dalla prefazione: "This book teaches how to write good programs that make good tools, by presenting a comprehensive set, each one of which provides lessons in design and implementation. The programs are not artificial, nor are they toys. Instead, they are tools which have proved valuable in the production of other programs. We use most of them every working day, and

they account for most of our computer usage. The programs are complete, not just algorithms and outlines, and they work: all have been tested directly from the text, which is in machine-readable form. They are readable: all are presented in a structured language called Ratfor (for "Rational Fortran"), which can be easily understood by anyone familiar with Fortran, PL/I, Cobol, Algol, Pascal or a similar language. (Ratfor translates readily into Fortran or PL/I: one of the tools presented is a preprocessor to translate Ratfor into Fortran automatically). [...] The book is pragmatic. We teach top-down design by walking through designs. We demonstrate structured programming with structured programs. We discuss efficiency and reliability in terms of actual tests carried out. [...] Indice: Introduction; 1. Getting started; 2. Filters; 3. Files; 4. Sorting; 5. Text Patterns; 6. Editing; 7. Formatting; 8. Macro Processing; 9. A Ratfor-Fortran Translator; Epilogue. (Tutti i programmi descritti nel libro sono disponibili in forma leggibile da calcolatore presso la Addison-Wesley).

Un libro sulla ricerca operativa

● Thesen, A., COMPUTER METHODS IN OPERATIONS RESEARCH, Academic Press, 1978, XIII+268.

"This text is designed to fill the growing gap between the computational requirements emerging in modern industrial engineering and operations research (IE/OR) applications and the algorithmic and computational methods commonly available to the IE/OR student and practitioner. [...] The academic level of a course based on this book would be suitable for a senior or first-year graduate student in engineering or business. Prerequisites for this course would include a course in FORTRAN programming and a course in deterministic OR models (including linear programming and network analysis). [...] " Indice: 1. Considerations in program design evaluation; 2. List processing; 3. Sorting and searching; 4. Networks-Fundamental concepts; 5. Critical Path Methods; 6. Resource constrained scheduling methods; 7. Linear programming methods; 8. Branch and Bound methodology; 9. Random number generators; 10. Discrete event simulation programming; 11. Two simulation languages (GPSS and WIDES).

Un'opera sulla storia del calcolo automatico

● Soresini, F., STORIA DEL CALCOLO AUTOMATICO, edizione fuori commercio a cura della Confederazione Generale dell'Industria Italiana, Roma, Ed. dell'Ateneo & Bizzarri, tre volumi: vol. I, CALCOLO NUMERICO E MECCANICO, pp. 178, vol. II, DALLA MECCANOGRAFIA ALL'INFORMATICA, pp. 276, vol. III, CALCOLO ANALOGICO E CIBERNETICA, pp. 212.

Un'ampia opera, corredata di un ricchissimo e raro repertorio iconografico, che consente di percorrere l'itinerario storico del calcolo meccanico, dall'antichità al periodo attuale.

Tesi di PhD

● Simonyi, C., META-PROGRAMMING: A SOFTWARE PRODUCTION METHOD, Stanford University, Department of Computer Science, PhD, dicembre 1976.

"[....] Meta-programming, the main subject of this dissertation, and its host organization, the Software Production Team (SPT) form an integrated method of software production. Just as in CPT, the SPT's first level manager, the meta-programmer, is directly involved in programming activities. [....] The most important feature of the SPT organization is the emphasis on optimizing productivity in the simpler phases of programming, which are detailed design (meta-programming), coding and debugging. [...] Difficulty of communication within a production unit, caused by the rapid creation of problem specific, local, language, is posited as the major obstacle to the improvement of productivity. [...] Within the SPT organization, language creation is carefully controlled

by the meta-programmer, who issues written meta-programs defining local language and specifying the program logic as well. Technicians write and debug the actual code on the basis of the meta-programs. A number of conventions for increasing the effectiveness of meta-programs, and for organizing the debugging activity are also presented. A detailed example is given to illustrate these ideas. The chapter concludes with a series of comparisons and possible combinations of SPT and other software engineering concepts, CPT in particular. Chapter 3 describes a series of experiments which were performed to measure the actual performance of the method. [...]"

● Thomas, J.W., MODULE INTERCONNECTION IN PROGRAMMING SYSTEMS SUPPORTING ABSTRACTION, Brown University, PhD, giugno 1976.

"In response to a widely recognized 'software crisis', research in software engineering has produced the promising new programming methodologies of 'structured programming' and 'information hiding'. These methodologies both depend upon abstraction in a crucial manner. A general framework for viewing abstractions is presented, and these methodologies are examined in that framework. New programming languages, such as CLU and ALPHARD, are being developed expressly to support these programming methodologies. The crucial concept in such languages is their notion of module - a unit of source code realizing some abstraction. This thesis is primarily concerned with controlling, establishing, and documenting the interconnections between such modules to form working systems. A language for specifying module interconnection in a machine checkable form is developed, and its incorporation into a programming system studied. A model of its implementation is provided. An organization is suggested for programming systems which incorporate both a language processor supporting 'modular abstractions' (e.g., CLU and ALPHARD) and a module interconnection language (MIL) processor for binding the modules together. The main characteristics of this organization are that i) it allows flexible bindings, and ii) it provides machine-enforced documentation of the entire module interconnection structure. In addition, the MIL can be used for structuring a library of predefined modules: a facility for encapsulating a set of modules and their interconnections called the subsystem is introduced. The MIL's binding scheme allows for very general sharing of both modules and subsystems."

● RIN, N.A., AUTOMATIC GENERATION OF BUSINESS DATA-PROCESSING PROGRAMS FROM A NON-PROCEDURAL LANGUAGE, University of Pennsylvania, PhD, 1976, Computer Science.

"[...] The literature on software development of the past several years has recognized the unlikelihood of continued building of large complex software systems by the conventional manual methods because of rising software development costs and demand on the one hand and the shortage of sufficient trained personnel on the other. This research and that of others (surveyed in Chapter 2) are demonstrating the feasibility of reducing the costs of software development by utilizing the computer itself to produce industrial software systems automatically based on functional specifications from the business or management specialist. [...] Toward this objective, the immediate goal of this dissertation has been the development of (1) a non procedural Module Specification Language (MODEL) in which an analyst specifies a functional module of an application system; and (2) a software system that automates a significant portion of the software development process; namely, the program module design, coding and debugging phases of a large class of conventional data processing programs to be described [...]"

● Carson, J.H., THE PRACTICAL IMPLEMENTATION OF STRUCTURED PROGRAMMING LANGUAGES, Leight University, PhD, 1976.

Vengono trattati sia i problemi di analisi sintattica (con l'aiuto di "augmented context free grammars"), sia di analisi semantica (per cui viene proposta una "limited attribute tree grammar"), sia di generazione di codice.

● Patterson, D.A., VERIFICATION OF MICROPROGRAMS, University of California Los Angeles, PhD, 1976.

"The thesis of this research is that high level language programming, structured programming and formal program verification can be combined to produce correct, efficient microprograms. Several tools were developed to test this thesis. A structured, high level microprogramming language STRUM was designed. An optimizing compiler was created that produces micropro - grams for a horizontally microprogram mable computer, the Burroughs D-Machine. A verification condition generator and algebraic simplifier were created to aid the formal verifica tion of microprograms[...]"

● Meehan, J.R., THE METANOVEL: WRITING STORIES BY COMPUTER, Yale University, PhD, dicembre 1976, Computer Science.

"The term metanovel, and the idea, is due to Alan Perlis. A metanovel is a computer program that tells stories that only a computer could tell, stories of such complexity of detail that only a computer could handle, stories with more flexibility - even reversibility - of events and characters than a human could manage. A metanovel time sharing system tells a story to many people at once, no two of whom read the same thing, because they have each expressed different interests in the events and characters they want to hear about, and because they may each desire a different style of storytelling. And yet, among all these readers, there is but one story - the Metanovel itself - and each reader is only following those threads of story that interest him.[...]"

Honeywell

Honeywell Information Systems Italia

Via Laboratori Olivetti - Pregnana Milanese

UNIVERSITA'
DEGLI STUDI DI MILANO

ISTITUTO DI CIBERNETICA

Via Viotti, 5 - Milano